2009/2010

# Adebug:

## Una Herramienta para la Depuración de Gramáticas de Atributos


**Nora Pacheco Blázquez**

Proyecto de Sistemas Informáticos
Facultad de Informática
Universidad Complutense de Madrid
**Director: José Luis Sierra Rodríguez**

*Quiero agradecer a José Luis toda la ayuda que me ha proporcionado para realizar este proyecto. Destacar su paciencia y comprensión.*

*A Antonio, por emplear parte de su tiempo en nuestras primeras reuniones.*

*A mis padres y amigos, por su apoyo y la confianza que han depositado en mí.*

A todos vosotros, gracias.

# *RESUMEN*

En este trabajo de Sistemas Informáticos se ha desarrollado un entorno gráfico de depuración para el procesamiento de gramáticas de atributos. Esta herramienta de depuración recibe cómo entrada una serie de sencillos comandos que es capaz de interpretar. Estos comandos pueden ser producidos desde diferentes fuentes, pero en nuestro caso hemos utilizado la herramienta BYacc. El depurador cuenta con una serie de modos distintos de ejecución que nos permiten movernos por el árbol de análisis atribuido de forma sencilla, rápida y sobre todo muy intuitiva. El crecimiento del árbol debido a la aplicación de reglas de producción, el establecimiento de las dependencias entre atributos heredados y sintetizados y las correspondientes evaluaciones de estos atributos son mostrados perfectamente en nuestra herramienta, pudiéndose elegir en cada momento una ejecución con o sin animación. De esta forma, conseguimos un correcto entendimiento de cómo estas gramáticas funcionan, lo cual, al fin y al cabo es el objetivo final.

**Palabras Clave:**

Herramienta de depuración, gramáticas de atributos, LALR(1), árbol de análisis sintáctico, grafo de dependencias, Piccolo, BYacc.

# *ABSTRACT*

In this project, a graphic debugging environment for processing of attribute grammars has been developed. This debugging tool has as input a list of simple commands which the environment is able to interpret. These commands can be produced from different sources, but in this case, the BYacc tool has been used. The debugging environment has different execution modes which allow us to move along the attribute parse tree in an easy, fast and especially very intuitive way. The growth of the tree due to the application of the production rules, the establishment of the dependences between inherited and synthesized attributes and the corresponding evaluations of these attributes are perfectly showed in the tool, being able to choose at each moment an execution with or without animation. In this way, a good understanding of how this kind of grammar works is got and this is, indeed, the final aim.

**Keywords:**

Debugging tool, attribute grammar, LALR(1), syntactic analysis tree, dependence graph, Piccolo, BYacc.

# INDICE

# 1.INTRODUCCIÓN

## 1.1 Presentación del problema

Entender correctamente cómo funciona una gramática de atributos es algo importante y no sólo desde el punto de vista académico, dónde forma parte del temario de la asignatura de Procesadores del Lenguaje. Actualmente, los archivos XML constituyen una pieza elemental en los escenarios de e-Learning y su procesamiento puede ser sistematizado mediante el uso de gramáticas de atributos. XLOP[1] es un entorno que las utiliza para procesar documentos XML. XLOP produce una implementación escrita en CUP del procesador asociado a una determinada gramática. CUP[2] es un sistema de generación de traductores para Java que soporta gramáticas LALR(1).

## 1.2 Objectivos del proyecto

El principal objetivo de este trabajo de Sistemas Informáticos es la construcción de un entorno gráfico de visualización para el procesamiento asociado con gramáticas de atributos que permita entender su correcto funcionamiento. Más concretamente se adopta como modelo de ejecución el que se utiliza en XLOP: construcción ascendente del árbol de análisis sintáctico, y evaluación de atributos dirigida por los datos (es decir, los valores de los atributos se evalúan en el momento en el que se dispone de los valores de todos los atributos de los que estos dependen). De esta forma, el entorno ayudará a entender cómo las reglas de producción son aplicadas y en qué orden esto se realiza, cómo las dependencias entre atributos asociados a nodos se van llevando a cabo y en qué momento y bajo qué circunstancias dichos atributos son evaluados.

La herramienta de depuración se llamará Adebug y recibirá una serie de comandos de ejecución escritos bajo un determinado criterio, siendo dichos comandos producidos por la implementación de la gramática a depurar, convenientemente instrumentada con instrucciones adicionales para generar los mismos. A partir de dichos comandos, que Adebug recibirá, la herramienta podrá llevar a cabo diferentes tipos de ejecución y pondrá a nuestra disposición una serie de utilidades que nos permitarán inspeccionar el árbol de análisis y el grafo resultante, así como seguir el proceso de evaluación de los atributos.

[1] Alberto Martinez, Bryan Temprado "XLOP - XML Language-Oriented Processing" (Proyecto fin de carrera, Universidad Complutense, Madrid, 2009)

[2] Appel, A.W. 1997. Modern Compiler Implementation in Java. Cambridge Univ. Press

Con el objetivo de probar la herramienta de depuración, se codificará una gramática de atributos mediante BYacc, siguiendo un patron de ejecución análogo al utilizado en XLOP, aunque más adecuado para la codificación manual de este tipo de gramáticas. Así mismo, anotaremos dichas implementaciones con instrucciones para la creación de los comandos que Adebug es capaz de interpretar.

Este trabajo servirá para poner en práctica los conocimientos adquiridos en la asignatura de Procesadores del Lenguaje y en las relacionadas con programación en Java. Igualmente se pretende adquirir nuevos conocimientos relacionados con tecnologías que eran desconocidas con anterioridad.

## *1.3 Estructura del proyecto*

Este documento está organizado en una serie de capítulos, estructurados de la siguiente forma:

- El capítulo 2 hace referencia a las tecnologías que han sido utilizadas, explicando de qué forma han sido útiles y utilizables en el contexto del proyecto. También aparece aquí una revisión de los conceptos necesarios para un correcto entendimiento de la memoria y, por conseguiente, del proyecto en general.
- En el capítulo 3 se explica cómo el depurador trabaja, los comandos que es capaz de interpretar y cómo éste lo hace. También se explican los diferentes modos de depuración que pueden ser encontrados, cómo estos funcionan y lo que esperamos como resultado en cada uno de los casos. Se explica también dentro de este apartado en qué consiste la interfaz del depurador y cómo usarla para obtener nuestros objetivos.
- El capítulo 4 incluye los detalles de implementación de la aplicación, cómo ésta se ha llevado a cabo de forma gradual y cómo está estructurada internamente. Se detalla, para ello, las clases que la componen y cómo se relacionan entre ellas.
- El capítulo 5 se centra en cómo codificar gramáticas de atributos en el sistema BYacc, y cómo anotar éstas para permitir la depuración con Adebug. Se detallan las transformaciones que sufre la gramática de entrada para obtener la correspondiente en lenguaje Byacc y cómo instrumentar ésta última para obtener los comandos que el depurador es capaz de interpretar. Aunque, de cara al presente proyecto, esta actividad se ha realizado manualmente, no sería difícil automatizarla, de forma parecida a como se hizo en el sistema XLOP.
- El último capítulo está destinado a exponer las conclusiónes de todo el trabajo realizado y las posibles ampliaciones y mejoras que pueden ser añadidas en el futuro al depurador.

# 2. REVISIÓN DE CONCEPTOS Y TECNOLOGÍAS

## 2.1 Gramáticas de atributos

Las gramáticas de atributos son una generalización de las gramáticas incontextuales cuyos símbolos pueden tener asociados atributos y cuyas producciones pueden tener asociadas reglas de evaluación. El término *incontextual* se refiere al hecho de que un no terminal puede siempre ser sustituido por el cuerpo de una producción asociada sin tener en cuenta el contexto en el que ocurra. El formalismo de las gramáticas de atributos fue propuesto por Donald E.Knuth con el objetivo de expresar la semántica de cualquier lenguaje incontextual[3].

Una gramática de atributos consistirá en:

- La **gramática incontextual** previamente mencionada que sentará la sintaxis del lenguaje. La síntaxis de la mayoría de lenguajes de programación está definida mediante gramáticas libres de contexto
- Los **atributos semánticos** que se asociarán a las categorías sintácticas.
- Las **ecuaciones semánticas** que servirán para calcular el valor de los atributos de las categorías sintácticas de las producciones usando los valores de otros atributos de los símbolos de la producción. Más concretamente, los atributos de una categoría sintáctica se dividen en atributos sintetizados y atributos heredados. Estas ecuaciones indican cómo computar los valores de los atributos sintetizados de la cabeza y de los atributos heredados de los símbolos del cuerpo.

Cada símbolo puede tener asociado un número finito de atributos. A su vez, cada producción puede tener asociada un número finito de reglas de evaluación de los atributos. Los valores de los atributos deberán estar asociados con un dominio de valores.

Como hemos indicado anteriormente, podemos encontrar dos tipos de atributos. Sea la regla:

$A \rightarrow A_1\ A_2 \ldots A_k$ con $a, a_1, a_2, \ldots, a_n$ atributos de los símbolos de la regla.

[3] Knuth, D.E. Semantics of Context-free Languages, Math.System Theory 2(2),127-145, 1968. See also Math.System Theory 5(1), 95-96. 1971

Los atributos puede ser:

- **Atributos sintetizados:**

  Su valor se calcula únicamente a partir de los valores de los atributos pertenecientes a sus hijos en el árbol de análisis.

  $A.a = f(A_1.a_1, A_2.a_2, \ldots, A_k.a_k)$

- **Atributos heredados:**

  Su valor se calcula a partir de los valores de los atributos pertenecientes al padre o a los hermanos de ese nodo en el árbol de análisis.

  $A_i.a_i = f(A_1.a_1, A_2.a_2, \ldots, A_{i-1}.a_{i-1}, A_{i+1}.ai_{i+1}, \ldots, A_k.a_k)$

- **Atributos léxicos:**

  Son los atributos sintetizados asociados a los nodos terminales de la gramática.

## 2.2 Grafos de dependencias

La mejor forma de representar las dependencias introducidas por las ecuaciones semánticas es mediante los llamados grafos de dependencias de atributos, donde los nodos representan las instancias de los atributos en los nodos de los árboles de análisis sintáctico y las fechas entre ellas representan las dependencias. Más concretamente, si el valor de 'a' se utiliza para calcular el valor de 'b', habrá un arco con origen en 'a' y con destino en 'b'.

A continuación podemos ver un ejemplo de una gramática donde todos los atributos asociados con los símbolos gramaticales son sintetizados. Estas gramáticas son conocidas como S-Atribuidas.

| | |
|---|---|
| *L* -> *E* | { L.val = E.val } |
| *E* -> *E* +*T* | { *E*0.val = *E*1.val + *T*.val } |
| *E* -> *T* | { *E*.val =*T*.val } |
| T -> T**F* | { *T*0.val = *T*1.val * *F*.val } |
| *T* -> *F* | { *T*.val = *F*.val } |
| F -> (*E*) | { *F*.val = *E*.val } |

*F -> dígito* { *F*.val = dígito.lex }

Dada dicha gramática, el árbol de análisis sintáctico asociado a la expresión *3*5+7*, junto con el grafo de dependencias entre los atributos de sus nodos, será:

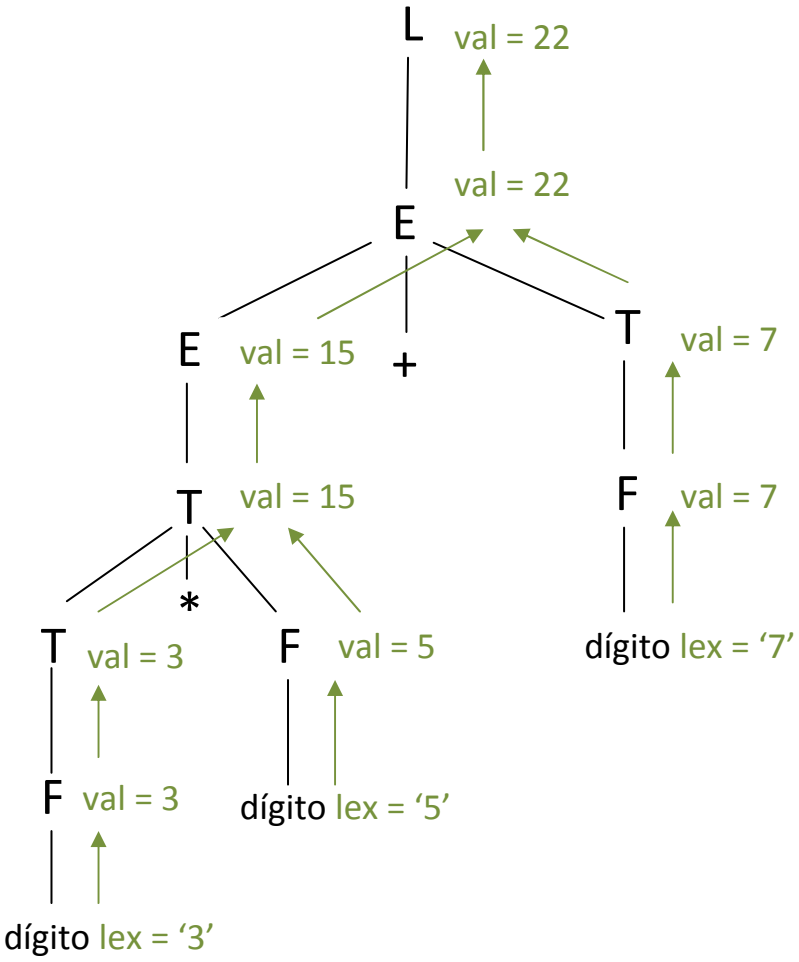


**Figura 2.2.1** Ejemplo de árbol de análisis y de grafo de dependencias asociado. El ejemplo involucra únicamente atributos sintetizados.

Como podemos observar, el valor de los atributos en cada nodo se calcula a partir de los valores de los hijos. Los atributos sintetizados de los terminales se denominan atributos léxicos (en nuestro ejemplo el atributo lex del terminal F), y su valor se asume recibido de manera externa a la gramática de atributos.

Para ver cómo se calculan los valores de los atributos, considérese el nodo situado en el extremo de la izquierda, que corresponde al uso de la producción *F -> dígito.* La regla semántica correspondiente, *F.val = digito.lex*, establece que el atributo *F.val* en el nodo tiene el valor 3 porque el valor de digito*.lex* en el hijo de este nodo es 3. De igual forma, consideramos la producción T -> *T* * *F*. El valor del atributo *T.val* en este nodo está definido por la siguiente ecuación semantica:

*T0.val = T1.val * F.val.*

Los números que acompañan a los no terminales sirven para identificar sus ocurrencias en las reglas, en caso de que aparezcan más de una vez. Aquí, *T1.val* tiene el valor 3 del hijo izquierdo y *F.val* el valor 5 del hijo derecho. Por tanto, *T0.val* adquiere el valor 15 en este nodo.

Al contrario de los atributos sintetizados, cuya evaluación únicamente depende de atributos en los nodos hijos de sus nodos, la evaluación de un atributo heredado depende de los atributos asociados con el nodo padre o con los nodos hermanos. El siguiente ejemplo es una gramática que tiene como objetivo asociar el tipo a cada variable en su declaración.

D -> T Lista {*Lista.tipoh = T.tipo*}

{D.decs = Lista.decs}

T -> entero {*T.tipo = entero*}

T -> real {*T.tipo = real*}

T -> bool {*T.tipo = bool*}

Lista -> Lista , ident

{*Lista1.tipoh = Lista0.tipoh*}

{Lista0.decs = *añadetipo (Lista1.decs, ident.lex, Lista0.tipoh)*}

Lista -> ident

{Lista.decs = *añadetipo (creaDecs(), ident.lex, Lista.tipoh)*}

La expresión *real var1, var2, var3* producirá el siguiente árbol atribuido:

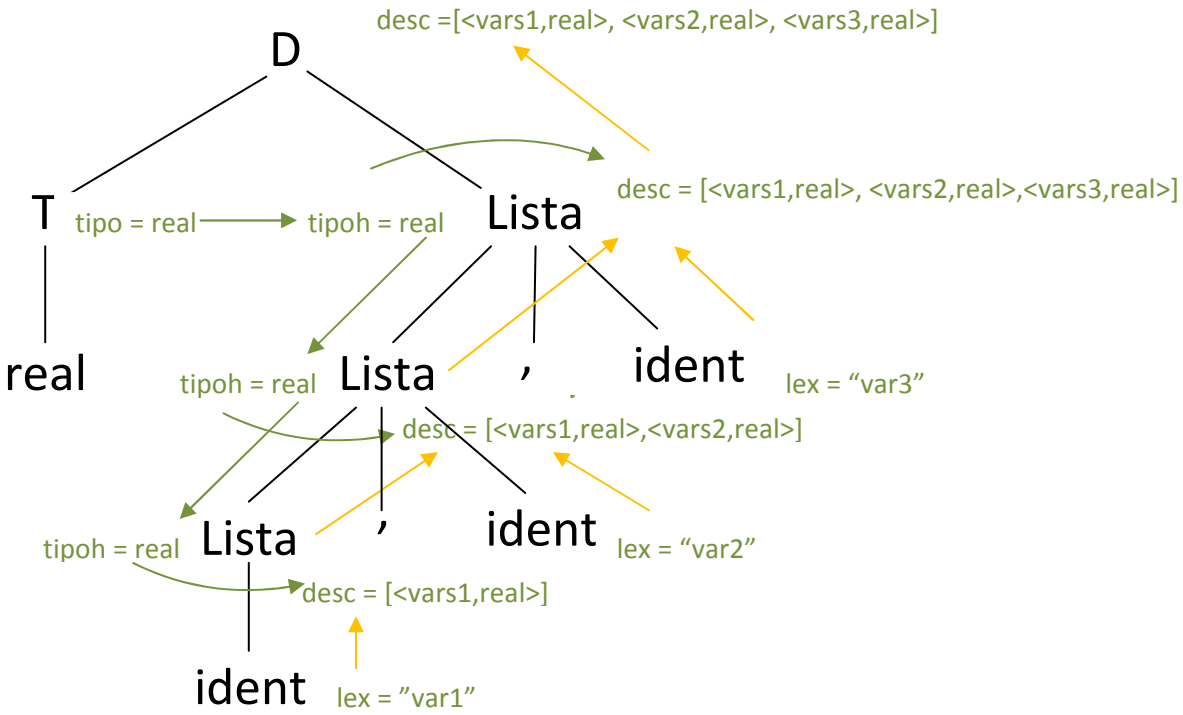


**Figura 2.2.2** Ejemplo de árbol de análisis y de grafo de dependencias asociado. El ejemplo involucra tanto atributos heredados como sintetizados.

Como se ha visto en estos ejemplos, las ecuaciones semánticas hacen uso de funciones semánticas (p.ej., añadetipo) que actuán sobre los atributos con el objetivo de calcular los valores de otros atributos.

## 2.2 Analizadores - Traductores

Según el orden en el que el analizador sintáctico construye los nodos del árbol, podemos hablar de dos tipos de analizadores sintácticos.

### 2.2.1 Analizador sintáctico descendente

Parte de la raíz del árbol (el símbolo inicial de la gramática S); aplicando las reglas de la gramática, sustituye la parte izquierda de una regla por su parte derecha y va generando el árbol sintáctico de arriba abajo, de la raíz a las hojas, que corresponden a los componentes léxicos básicos, los componentes léxicos (o tokens) de la cadena de entrada.

### 2.2.2 Analizador sintáctico ascendente

Parte de la frase que hay que analizar transformada en tokens por el analizador léxico, que constituirán los símbolos terminales del árbol sintáctico (las hojas). Va aplicando en sentido inverso las reglas de la gramática para crear los nodos intermedios del árbol, reduciendo las partes derechas de las producciones por las partes izquierdas. De esta manera se sube por los nodos del árbol hasta llegar a la raíz, el símbolo inicial de la gramática o

axioma. El árbol se construye, por lo tanto, de abajo hacia arriba, de las hojas a la raíz. La herramienta de depuracion Adebug se basa en este tipo de análisis.

En general, el análisis ascendente es más potente que el descendente (por ejemplo, el análisis ascendente puede manejar gramáticas recursivas a izquierdas, que causan problemas en el análisis descendente).

### *2.2.3 Funcionamiento de un analizador ascendente*

El analizador ascendente se encarga de encontrar subcadenas de la entrada que sean la parte derecha de una producción y la sustituye por la parte izquierda correspondiente. Estas subcadenas son llamadas asideros y en su búsqueda está basada el funcionamiento del analizador. Veamos un ejemplo. Dada la siguiente gramática:

S -> aAAeBd

A -> bB|b

B -> d

Supóngase que recibimos como entrada la cadena: abdbbdd

Durante el proceso de análisis ascendente, aplicamos las siguientes reducciones:

| | | |
|---|---|---|
| A -> b | --------> | aAbdedd |
| B -> d | --------> | aAbBedd |
| A -> bB | --------> | aAAedd |
| B -> d | --------> | aAAeBd |
| S -> aAAeBd | --------> | FIN |

El problema que aquí encontramos es que aparecen varias subcadenas en la parte derecha que pueden ser reducidas mediante reglas de la gramática. Obviamente no es indiferente el orden en el que estas reducciones se apliquen. Dependiendo de ello, podríamos llegar hasta el axioma y procesar la entrada correspondiente o llegar a una cadena que no contiene ninguna subcadena que pueda ser reducida llegando por tanto a un estado de bloqueo. Con el objetivo de solucionar esto, han aparecido dos técnicas diferentes: analizadores con retroceso y analizadores predictivos.

#### *2.2.3.1 Analizadores con retroceso*

Prueban todas las posibilidades con un algoritmo de backtracking. Es posible que tras llevar a cabo una acción semántica, necesitemos retroceder porque nos hayamos dado cuenta de que no era la correcta. Esto nos obliga a llevar a cabo, por un lado, la fase de análisis y una vez finalizada ésta, en una etapa posterior, la ejecución de las ecuaciones podría ser realizada.

### *2.2.3.2 Analizadores predictivos*

Utilizan información de la cadena de entrada para predecir qué subcadena lleva al éxito. Dependiendo de sus caracteristicas podemos clasificarlos en diferentes grupos (LR(K), SLR(1), LALR(1)). Prestaremos especial atencion a los analizadores LALR(1) por ser el tipo de analizadores que utilizan los parsers escritos en Java generados por el sistema CUP.

### *2.2.3.3. Gramáticas LALR(1)*

Las gramáticas LALR(1) [Look-Ahead LR] son las más expresivas para las que es posible generar analizadores compactos y eficientes. Los tipos de analizadores que trabajan sobre este tipo de gramática se basan en la anticipación en cada uno de los tokes de la cadena de entrada con el objetivo de tomar una correcta decisión en la selección de la siguiente regla de la gramática a ser aplicada. La letra L representa la dirección en la que procesamos la cadena de entrada, en este caso de izquierda a derecha, mientras que la letra R representa qué dirección tiene la derivación, en este caso por la derecha. El método LALR(1) surgió derivado del LR con el objetivo de disminuir el número de estados que éste último producía simplificando el análisis. El método de construcción de una tabla de análisis LR es más potente (abarcaba un número de gramáticas mayor) pero a cambio, también consume mucho más espacio.

Teniendo un gramática LALR(1), un analizador LALR(1) es capaz de saber en todo momento qué regla aplicar sin margen de error y por tanto llevar a cabo las acciones asociadas de forma segura.

#### 2.2.3.3.1 Automáta LALR

Como todos los analizadores LR, los basados en el método LALR(1) utilizan un automata finito para decidir qué acciones realizar durante el proceso de análisis. En el caso de las gramáticas LALR(1), dicho automata se denominará automata LALR. Su naturaleza se ilustrará con un ejemplo.

Se va a construir el automáta LALR asociado a una determinada gramática que permite gobernar el proceso de análisis de cualquier cadena de entrada que se rige de acuerdo a ella.

Partimos de la siguiente gramática:

S -> A

A -> BB

B -> cB

B -> d

Si no tenemos en cuenta los colores, el autómata obtenido (ver figura 2.2.3) es un automáta LR(1). Cada estado consisten en una serie de elementos, cada uno de ellos formado por una posición en la gramática (una producción con un punto en el lado

derecho), y por un símbolo de anticipación. La ventaja de los autómatas LALR(1), como ya hemos comentado anteriormente, consiste en la reducción en el número de estados con respecto a los LR(1) con el objetivo de simplificar la fase de análisis. Como podemos ver en el dibujo, existen cuatro estados (7,8,9,10) que pueden ser eliminados, ya que los estados 7 y 9 son fusionables con el 3 y, respectivamente, el 8 y el 10 lo son con el 4. Decimos que dos estados son fusionables cuando involucran exactamente a las mismas posiciones. En el autómata LALR(1), los estados fusionables se juntan en uno único. Los símbolos de anticipación de cada posición se unen, y las transiciones se matienen.

Podemos ver el resultado en la Figura 2.2.4

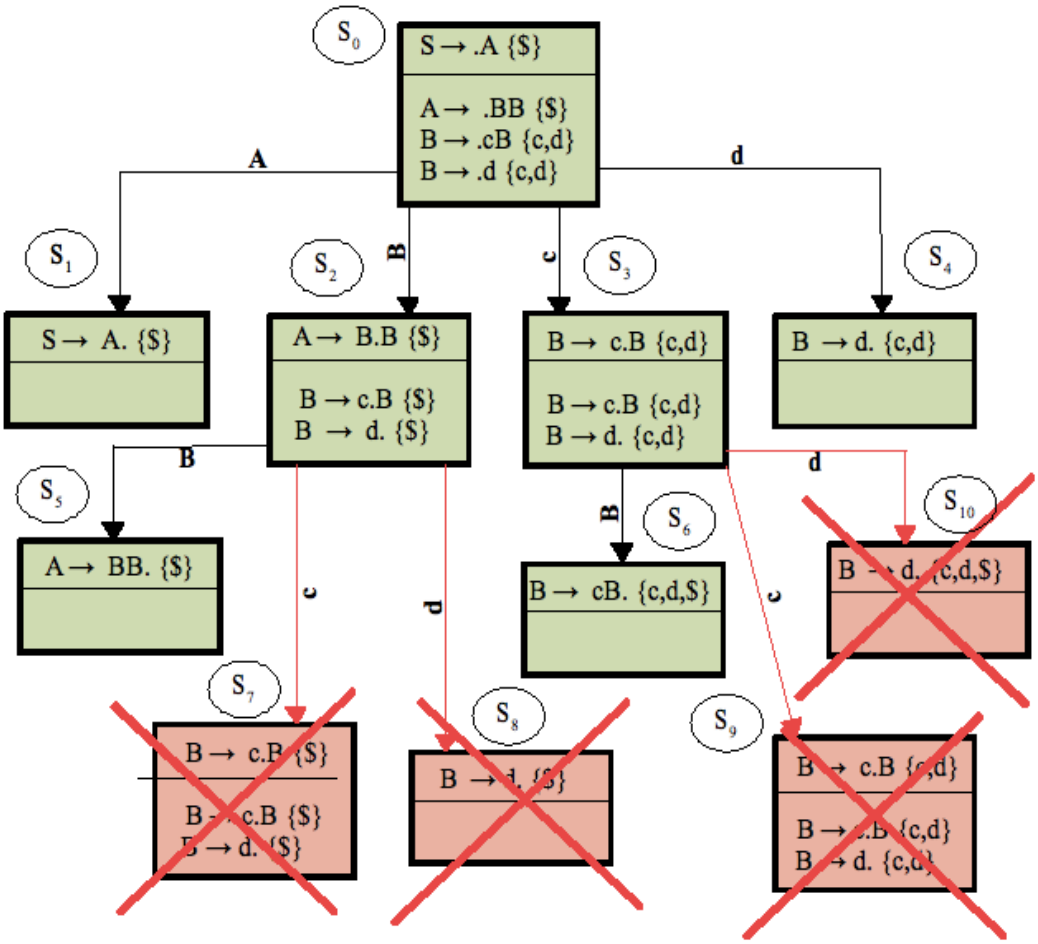


**Figura 2.2.3** Autómata LR(1)

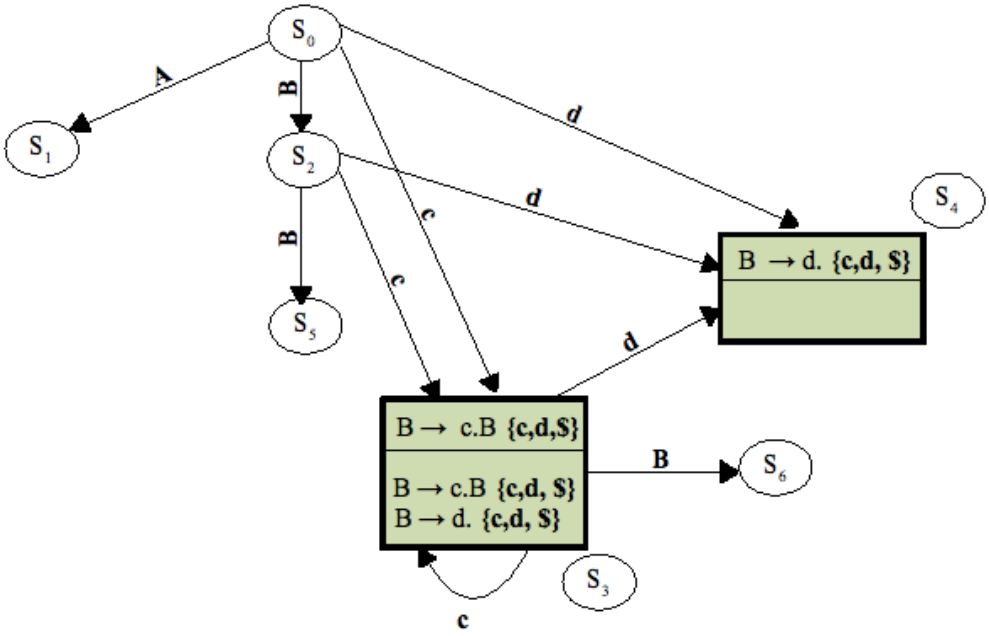


**Figura 2.2.4** Autómata LALR(1)

### 2.2.3.3.2. Tablas de análisis LALR(1)

Para llevar a cabo el análisis, debemos construir una tabla basándonos en el autómata LALR previamente construido. El analizador utilizará esta tabla para saber qué acción aplicar dependiendo del estado en el que se encuentre, así como del símbolo actual de la entrada. Para nuestro ejemplo, la tabla se muestra en la Figura 2.2.5.

| | *c* | *d* | *$* | *A* | *B* |
|---|---|---|---|---|---|
| *0* | d3 | d4 | | 1 | 2 |
| *1* | | | Aceptar | | |
| *2* | d3 | d4 | | | 5 |
| *3* | d3 | d4 | | | 6 |
| *4* | r3 | r3 | r3 | | |
| *5* | | | r1 | | |
| *6* | r2 | r2 | r2 | | |

**Figura 2.2.5** Tabla de análisis LALR(1)

Se mostrará el funcionamiento mediante un ejemplo. La cadena de entrada que nuestro analizador va a procesar es "dd$". El analizador se sirve de una pila de análisis y realiza una secuencia de desplazamientos y reducciones de acuerdo a las instrucciones

codificadas en la tabla (veáse la Figura 2.2.6.) y cuyos pasos seguidos son explicados posteriormente.

| | *Pila* | *Entrada* | *Acciones* |
|---|---|---|---|
| *0* | 0 | d d $ | desplaza(d), estado 4 |
| *1* | 4 d 0 | d $ | reduce(B -> d), estado 2 |
| *2* | 2 B 0 | d $ | desplaza(d), estado 4 |
| *3* | 4 d 2 B 0 | $ | reduce(B -> d), estado 5 |
| *4* | 5 B B 0 | $ | Reduce (A->BB), estado 1 |
| *5* | 1A 0 | $ | Reduce (S -> A $) |
| *6* | S0 | | Aceptar |

**Figura 2.2.6** Funcionamiento del analizador

Los pasos a seguir son los siguientes:

1. Introducimos en la pila el estado inicial (estado 0).
2. Se busca en la tabla la instrucción correspondiente a estar en el estado 0 y recibir el símbolo 'd'. Apilamos en la pila el nuevo estado tras haber apilado el símbolo de la entrada que ha sido procesado y, por tanto, también eliminado de la cadena de entrada. En ese estado se aplica la reducción B -> d, que provocará la siguiente actualización de la pila: se desapilan dos elementos (el doble de la longitud del cuerpo de la producción), se indexa la tabla con el estado que aparece en la cima y con la cabeza de la producción (en este caso, posición 0 y B, que vale 2), se apila la cabeza de la producción (B) y, por último, el estado resultante de la anterior indexación (2).
3. Estando en el estado 2 con el símbolo 'd' como símbolo actual, la posición (2,d) de la tabla nos dice que hay que desplazar el valor 'd' a la pila y transistar al estado 4, el cual también tiene que ser apilado.
4. La entrada (4,$) de la tabla indica que se debe reducir por B -> d, de modo que se desapilan dos elementos de la pila, descubriendo el estado 2. La entrada (2,B) indica que debe transitarse al estado 5, por lo que se apila B, y luego 5.

5. En el estado 5, y con $ en la entrada, la tabla nos indica que reduzcamos por A->BB. Desapilamos cuatro símbolos, descubriendo el estado 0. La entrada (0,A) nos indica que transitemos al estado 1, por lo que apilamos A, y luego 1.

6. Aplicación de la última reducción, obteniendo como cima de la pila el axioma de nuestra gramática y por consiguiente el fin de nuestro análisis.

## 2.3 Marcos de desarrollo

### 2.3.1 Piccolo

#### 2.3.1.1.Introducción

Piccolo[4] es una herramienta de visualización que nos ofrece la forma de crear complejas aplicaciones gráficas en Java. Posee una serie de llamativos efectos entre los que debemos destacar la animación, las representaciones múltiples y, especialmente, el que es su punto fuerte: las interfaces de usuario ampliables (Zoomable User Interface). La ZUI es un tipo de interfaz que presenta un canvas de información que permite al usuario aumentar o disminuir de forma continuada el canvas con el objetivo de obtener una información más detallada o una visión general de los elementos que forman parte de él.

Piccolo contiene una estructura jerárquica de objetos y cámaras, permitiendo al desarrollador de la aplicación orientar, agrupar o manipular grupos de objetos, lo cual ofrece una gran flexibilidad de visualización. Por ejemplo, si se aplica cierta propiedad a un componente padre, esa propiedad será heredada por todos los componentes hijos asociados al padre.

Otro de los aspectos que deben ser mencionados, es la facilidad que ofrece Piccolo de manipular los componentes que ya han sido dibujados. Para ello Piccolo cuenta con las llamadas "actividades". Estas actividades, que pueden ser totalmente manipuladas por el desarrollador, son utilizadas para dar vida a la interfaz a través del uso de la animación y de comportamientos programados. Controlan los aspectos dependientes del tiempo de Piccolo y su duración puede ser fijada de antemano o puede hacerse depender de alguna condición de terminación.

La interación con el usuario también es muy importante. Piccolo cuenta con una serie de "event listeners", a través de los cuales, se puede definir los distintos modos de

[4] Bederson, B.M., Grosjean, J., Meyer, J., Toolkit Desing for Interactive Structured Graphics, *IEEE Transactions on Software Engineering,* 30 (8), 535-546. 2004

interación entre usuario e interfaz. Su diseño, es por tanto, una parte principal de la creación de interfaces con Piccolo.

Piccolo permite por tanto, construir aplicaciones gráficas sin preocuparnos de los detalles de bajo nivel. Esta infraestructura nos suministra una forma eficiente de dibujar por pantalla, de manejar los limites de los elementos, manejo de eventos y una serie de facilidades que ayudan a controlar el uso del ratón sobre nuestro canvas.

Las diferentes versiones de Piccolo pueden ser descargadas y utilizadas libremente ya que es un software de código abierto.

Podemos encontrar tres diferentes versiones:

- **Piccolo.Java**

  Está escrito 100% en java y por tanto puede ser ejecutado en diferentes máquinas incluyendo Windows, Mac OS X, Linux y Solaris. Está basada en el API Java2D.

- **Piccolo.NET**

  Está escrito 100% en C# y puede ser ejecutado en plataformas que suministran el Framework .NET (básicamente las maquinas Windows). Suministra la misma funcionalidad que Piccolo.java, permitiendo a los usuarios .NET integrar facilmente gráficos 2D y efectos de zoom. Piccolo.NET está basado en el API GDI+.

- **PocketPiccolo.NET**

  Al igual que Piccolo.NET está escrito en C#. Puede ser ejecutado en plataformas que soporten el framework Compact.NET. Esto significa, que está principalmente dirigido para PDAs que corran el sistema Pocket PC, incluyendo algunos modelos de teléfonos móviles.

#### *2.3.1.2 Origen*

La versión Java de Piccolo fue creada por Jesse Grosjean y posteriormente portada a .NET por Aaron Clamage, bajo la dirección de Ben Bederson en la Universidad de Maryland. Piccolo surgió como consecuencia de lo que ellos habían aprendido años atrás (Pad++ y Jazz fueron sus anteriores proyectos) y con el objetivo de obtener un pequeño motor que fuera sencillo y fácil de entender, usar y extender.

#### *2.3.1.3 Diseño*

El framework Piccolo está compuesto por una serie de clases principales:

- **PNode**

Los nodos son el concepto de diseño central en Piccolo. Cualquier objeto que quiera ser pintando en la pantalla debe heredar de la clase PNode. Estos nodos a su vez pueden tener asociados una serie de hijos que serán a su vez nodos. Las estructuras visuales son estructuras creadas por agrupaciones entre ellos. Si aplicamos una transformación a alguno de estos nodos, ésta tendrá efecto en los nodos hijos pero no en el nodo padre.

- **PLayer**

Son nodos que pueden ser vistos por una o más cámaras. Poseen la lista de cámaras que les ven con el objetivo de notificarles cuándo tienen que ser actualizadas.

- **PCamera**

Las cámaras son nodos que tiene una "additional view transform" y una colección de PLayers y otra de los hijos que heredan de PNode. Podrían referenciar a Pcanvas con el objetivo de representar los eventos.

- **PRoot**

Es el nodo más alto en la jerarquía. El resto de nodos, o son sus hijos directos, o descendientes de sus hijos. El PCanvas se comunica con el nodo root para manejar las actualizaciones de pantalla y enviar los eventos a sus hijos.

- **PCanvas**

Es un JComponent en Piccolo.Java y un Control en Piccolo.NET. Por tanto, es usado para mostrar el grafo de escena de Piccolo en Java Swing y aplicaciones Windows .NET respectivamente. El PCanvas muestra el grafo de escena a través de una PCamera. Dirige los eventos de entrada hacia esa cámara y usa la cámara en sí misma para dibujarlos. Trasladando y escalando la vista de la cámara se logran tanto el barrido como el zoom. Podemos ver en forma de árbol su relación en la siguiente figura:

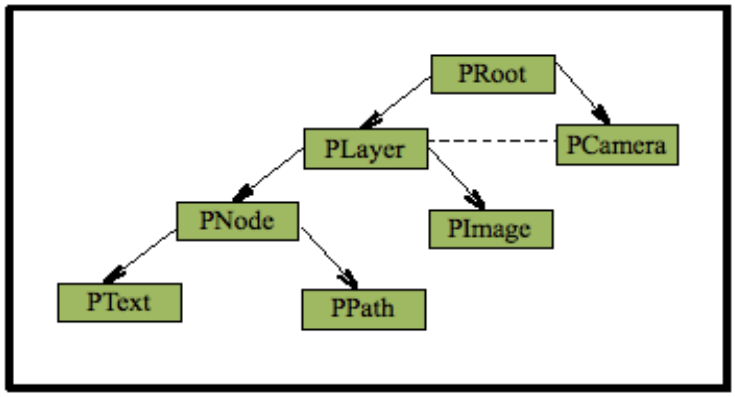


**Figura 2.3.1** Relación entre las clases principales de Piccolo

El significado de las fechas en esta figura no es el de la herencia. Exceptuando PText y PPath que sí heredan de PNode, el resto se relacionan entre ellas mediante la llamada al método *addChild* (veáse Figura 2.3.2).

```
PLayer edgeLayer = new PLayer();
getRoot().addChild(edgeLayer);
getCamera().addLayer(0, edgeLayer);

public void dibujaFlecha2(PPath line) {
    PLayer nodeLayer = this.getLayer();
    nodeLayer.addChild(line);
    nodeLayer.addChild(triangulo);
}
```

**Figura 2.3.2.** Se muestra cómo se añade el *layer* al *root* y cómo posteriormente se añaden dos elementos de tipo PNode al propio layer. El elemento triángulo que aparece en la figura corresponde a la clase PPath al igual que *line*.

### *2.3.1.4 Piccolo en Adebug*

Con el objetivo de dibujar el grafo de escena que nos suministra Piccolo se ha creado la clase *Grafo.* En ella encontraremos toda la parte del depurador relacionada con Piccolo. Esta clase hereda de PCanvas, que, como se ha comentado anteriormente, es necesaria para mostrarlo. Una instancia de esta clase Grafo será añadida al contenedor de un JInternalFrame (*PantallaGrafo*) que forma parte de la ventana principal de nuestro depurador.

Como ya se ha mencionado una de las principales ventajas de Piccolo, es la ZUI (Zoomable User Interface; ver Figura 2.3.3). Esta viene incorporada por defecto en Piccolo, y la forma de observar el zoom es clickeando prolongadamente con el botón derecho del ratón sobre el grafo de escena suministrado por Piccolo, el cual nosotros vemos dentro del JInternalFrame.

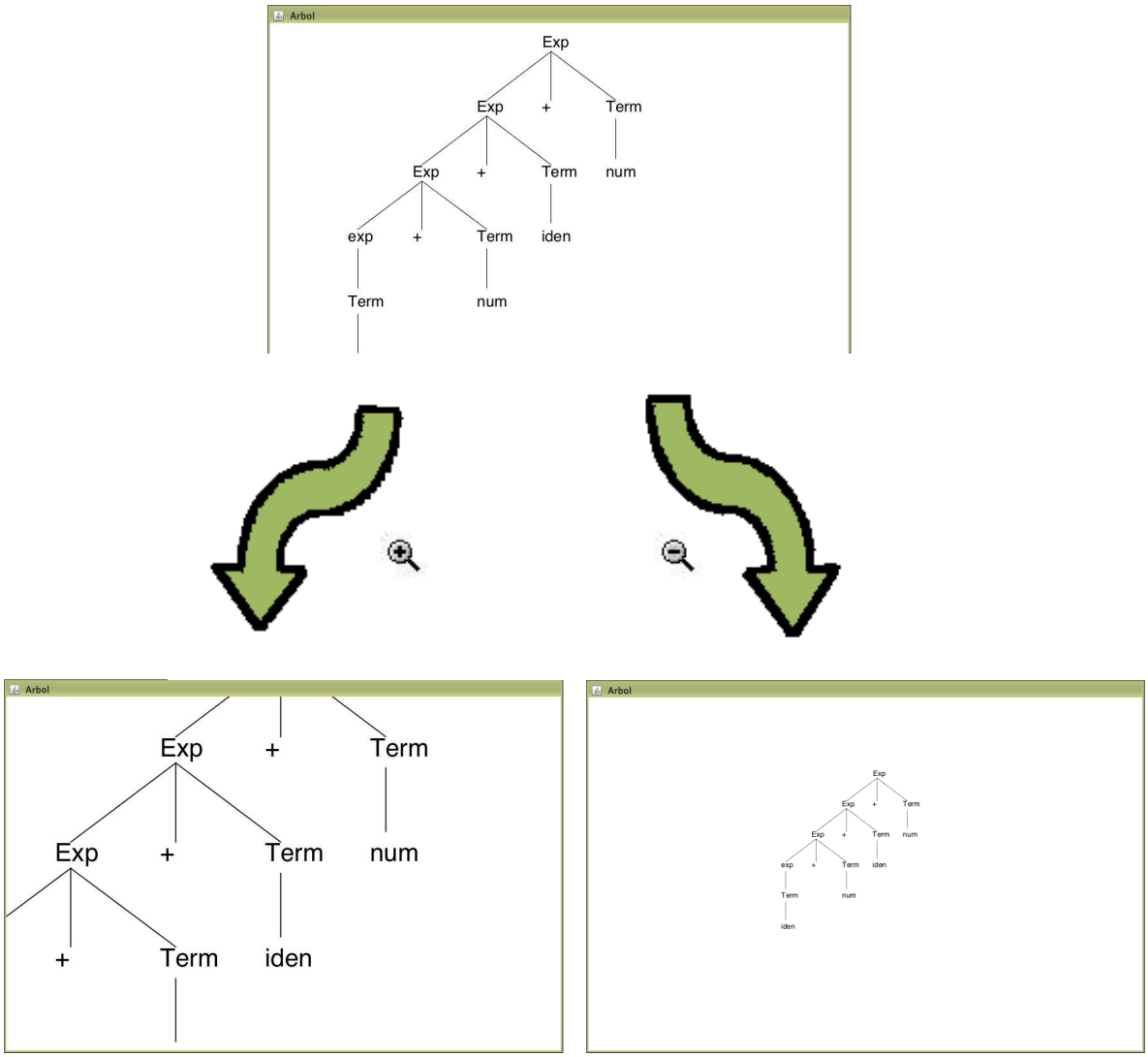


**Figura 2.3.3.** Imágenes de la aplicación del zoom.

Cada uno de los objetos que se muestran en el grafo son instancias de PNode (PText en el caso de los campos de texto y PPath para las lineas y flechas ). En ambos casos, los objetos son tratados como nodos que representan los componentes discretos de la interfaz. Estos nodos pueden ser animados con el objetivo de moverlos por el Canvas, mediante el control de las anteriormente mencionadas actividades (veáse Figura 2.3.4).

```
t.animateToPositionScaleRotation(p.dameAncho(), p.dameAlto(), t.getScale(), 0, 2000);
```

**Figura 2.3.4.** Método utilizado para trasladar los elementos. Como se puede observar, elegimos el tiempo en el que queremos que la animación se realice y el tamaño del item al final de ella.

### 2.3.2 Substance

Substance es un tipo de librería "Look & Feel" destinada a aplicaciones hechas con Java y cuyo objetivo es cambiar la apariencia visual de las interfaces gráficas de usuario. Las librerías Look & Feel nos permiten obtener un estilo más profesional, moderno o atractivo en nuestras aplicaciones cambiando el aspecto por defecto que nos proporciona el Swing de Java. Su principal ventaja es que esto puede hacerse fácilmente añadiendo sólo unas lineas al código a la aplicación. En la Figura 2.3.5 vemos cómo se ha aplicado Substance al depurador.

```
public PantallaSeleccionGramatica() {

    try {
        SubstanceLookAndFeel.setSkin("org.jvnet.substance.skin.SaharaSkin");
        initComponents();
        substance();
        this.setLocation(300, 250);
        this.setVisible(true);
        this.setResizable(false);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

**Figura 2.3.5.** Constructora de la clase PantallaSeleccionGramatica (la primera que se muestra en la aplicación). El método substance() cambia el diseño por defecto de los botones usando Substance, dotándoles de una forma redondeada.

El establecimiento del nuevo "Look & Feel" se realiza una vez y se mantiene durante la ejecución de la aplicación.

Substance destaca de otras librerías "Look & Feel" porque contiene un gran número de diferentes *Skins* que pueden ser utilizadas y porque nos permite modificar con mucha facilidad el estilo de los elementos del Swing (veáse, por ejemplo, la Figura 2.3.6 y la Figura 2.3.7)

```
buttonSiguiente.putClientProperty( SubstanceLookAndFeel.BUTTON_SHAPER_PROPERTY, new StandardButtonShaper());
```

**Figura 2.3.6.** Establecimiento de forma redondeada en el botón "Siguiente".

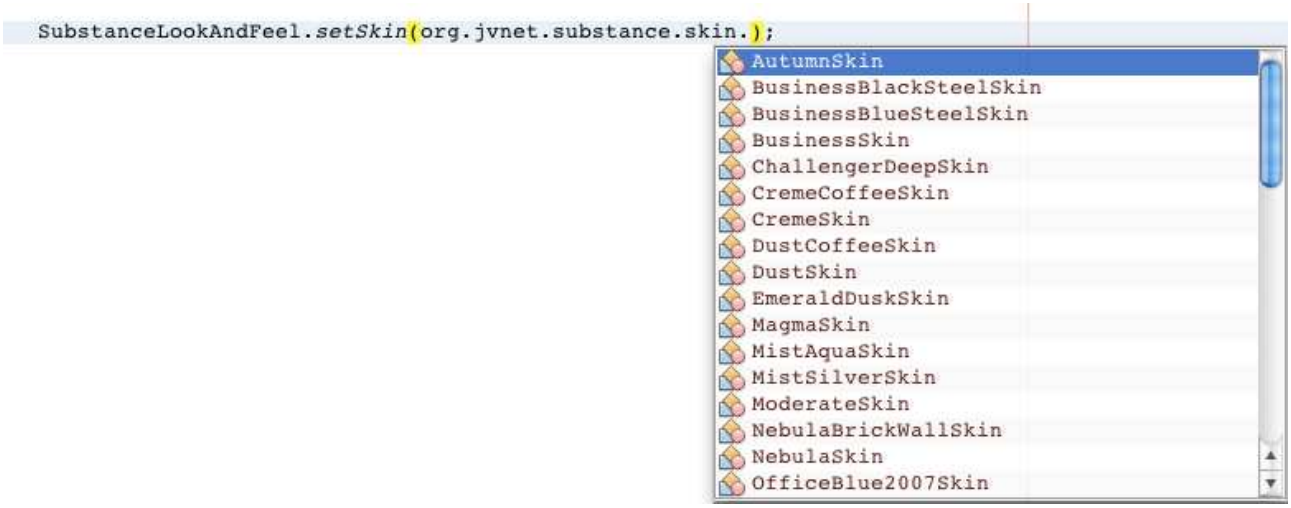


**Figura 2.3.7.** Visualización de las diferentes Skins que forman parte de Substance

# 3. EL SISTEMA ADEBUG

## 3.1 Introducción

El depurador cuenta con una serie de acciones que pueden ser llevadas a cabo a partir de unos comandos de entrada que son suministrados. Estas acciones se encargarán de trabajar con estos comandos consiguiendo así una correcta interpretación de la aplicación de las producciones de la gramática y de las acciones semánticas asociadas a ellas.

Contamos con formas de avance hacia delante y hacia atrás, ésta última con el objetivo de deshacer alguna acción previamente realizada sin necesidad de comenzar de nuevo, facilitándonos así concentrarnos en un determinado punto del desarrollo del árbol asociado sin tener que realizar de nuevo acciones necesarias para llegar hasta dicho punto.

El depurador consta de una interfaz muy intuitiva y fácil de utilizar. A continuación se explicarán las pantallas de las que consta, su funcionamiento y cómo por medio de ella podemos llevar a cabo cada una de las acciones de las que dispone el depurador.

## 3.2. Pantalla Inicial (Selección de gramática)

Cuando el depurador es ejecutado, lo primero que tenemos que indicar es el nombre de la clase java que nos suministrará la gramática a depurar, convenientemente instrumentada (la cual puede haber sido obtenida, por ejemplo, mediante BYacc). El nombre de esta clase debe ser escrito sin incluir la extensión de la clase.

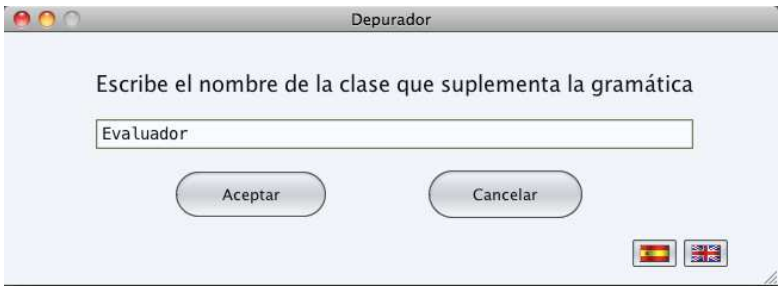


**Figura 3.2.1.** Pantalla de selección de gramática

## 3.3 Pantalla Principal

A continuación encontramos explicados los diferentes apartados de los que se compone la pantalla principal del depurador.

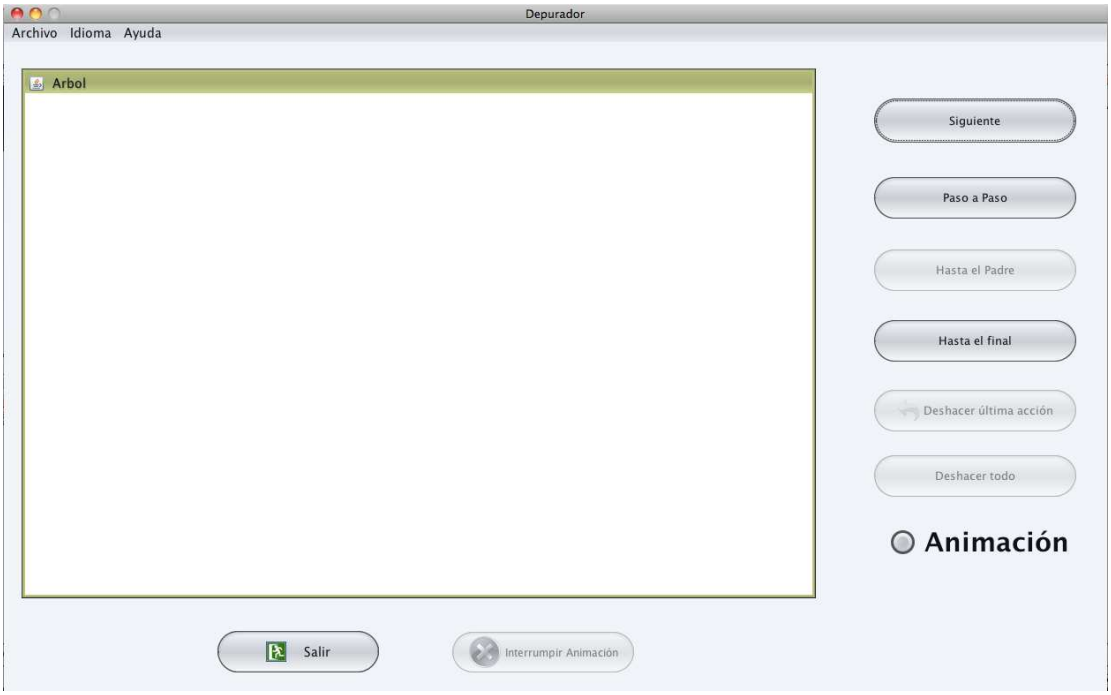


**Figura 3.3.1.** Pantalla principal de Adebug

### 3.3.1 Menú Horizontal

Consta de las siguientes opciones:

- Archivo

  Bajo esta pestaña, podrá seleccionar la opción de salir de la aplicación.

- Idioma

  Se podrá cambiar el idioma (inglés o castellano) en cualquier momento de la ejecución de la herramienta.

- Ayuda

  En caso de dudas con respecto al manejo de algunas de las funciones de la aplicación, aquí se podrá el usuario remitir para solucionarlas.

### *3.3.2 Animación*

Como se puede observar en la pantalla principal, tenemos la opción de seleccionar la ejecución con animación o no hacerlo. Según esta elección, visualmente, los resultados obtenidos en la ejecución serán diferentes y con respecto al estilo de visualización especialmente notables según la acción que internamente se esté ejecutando en cada momento. Las diferencias que encontraremos serán las siguientes:

- La aparición de nuevos nodos en caso de haberse producido un Reduce, un Shift, o agrupaciones de ellos, se hará de forma continuada en caso de haber seleccionado la ejecución con animación. Es decir, el árbol que vemos en ese momento se desplazará por el canvas hasta situarse en su nueva posición a medida que los nuevos nodos aparecen igualmente desplazándose de forma continuada. Si no se ha seleccionado la animación, el grafo anterior a la ejecución se eliminará del canvas y aparecerá el nuevo grafo directamente, sin ningún tipo de movimiento intermedio entre ambos.

- Si la animación no es seleccionada, cada vez que el depurador observe una evaluación de un atributo o el establecimiento de una dependencia, no mostrará tales acciones, exceptuando aquel momento en el que se haya seleccionado el modo "Avanzar Paso a Paso".

En general, con la activación de la animación entenderemos mejor cómo los atributos de los nodos del árbol se van evaluando y cómo están relacionados entre sí. Sin embargo, el tiempo empleado en la visualización del árbol será notablemente mayor y esto a veces puede no ser deseado.

### *3.3.3. Botones*

El depurador cuenta con una serie de botones a través de los cuáles las diferentes modos de ejecución serán llamados.

- Botón "Siguiente"

La ejecución del árbol se producirá hasta la siguiente acción de shift o reduce, por lo que visualmente, el efecto por tanto que obtendremos será de un crecimiento del árbol siempre que presionemos el botón.

- Botón "Paso a paso"

Cada vez que pulsemos este botón, el nuevo árbol aparecerá de forma continuada, desplazándose por el canvas si así es requerido. La diferencia con el botón Siguiente es que cuando un atributo sea evaluado se mostrará su valor y las correspondientes flechas que mostrarán las dependencias entre los atributos relacionados con el que se acaba de evaluar.

- Botón "Hasta el Padre":

La ejecución avanzará hasta el padre del último nodo que ha sido añadido al árbol. Si la ejecución no ha comenzado, este modo de ejecución no estará disponible.

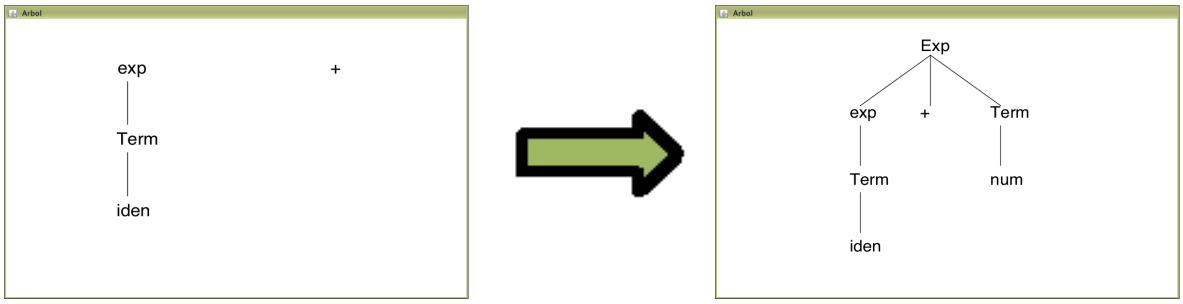


**Figura 3.3.2.** Ejemplo de ejecución modo "hasta el padre".

- Botón "Deshacer Ultima Acción"

Deberemos de seleccionar esta acción cuando queramos deshacer la última acción llevada a cabo tras haber elegido alguno de los modos de ejecución. Es decir, si la acción ha sido avanzar hasta el padre, se deshará la acción, obteniendo el árbol que había antes de haberla seleccionado, e igualmente con el resto de acciones. Este botón funciona a modo de pila. Es decir, podremos deshacer todos los movimientos desde el comienzo de la ejecución. Todas las acciones serán deshechas excepto la asociada con la obtención de un nuevo grafo.

- Botón "Hasta el Final"

Con este botón, la ejecución comenzará desde el punto en el que se encuentre y no parará hasta llegar a la cima del árbol, es decir, la cabeza de la producción más externa.

- Botón "Deshacer Todo"

Si pulsamos esta opción, se borrará el grafo que tenemos y la aplicación estará dispuesta para empezar de nuevo otra ejecución.

- Botón "Salir":

Cierra la aplicación.

- Botón "Interrumpir Animación":

Este botón puede ser pulsado cuando la animación esté activada y se haya seleccionado alguno de los botones de avance exceptuando el modo "Paso a Paso".

### *3.3.4 Grafo*

Dentro de la pantalla principal de la aplicación, podemos encontrar un JInternalFrame donde el grafo deseado aparecerá y se desarrollará según avance la ejecución.

En este JInternalFrame podremos encontrar todas las facilidades que Piccolo nos da (ver apartado 2.3.1). Gracias a ello, un nuevo modo de ejecución puede ser llamado. Si se clickea sobre alguno de los nodos del árbol se podrá seleccionar alguno de los atributos que forman parte del nodo y se podrá avanzar la ejecución hasta la evaluación de dicho atributo previamente seleccionado.

#### *3.3.4.1 Ejecución hasta la evaluación de un atributo*

Contamos con este modo de ejecución para llevar a cabo el desarrollo del árbol hasta el momento en el que se conozca el valor de un atributo asociado a un nodo. Para obtener esto, tenemos que proceder como sigue:

**1)** Clickear sobre el nodo cuyo atributo asociado queremos conocer el valor.

**2)** Nos aparecerá una pantalla donde se mostrará una lista con los atributos asociados al nodo sobre el que previamente hemos clickeado. Veáse Figura 3.3.3.

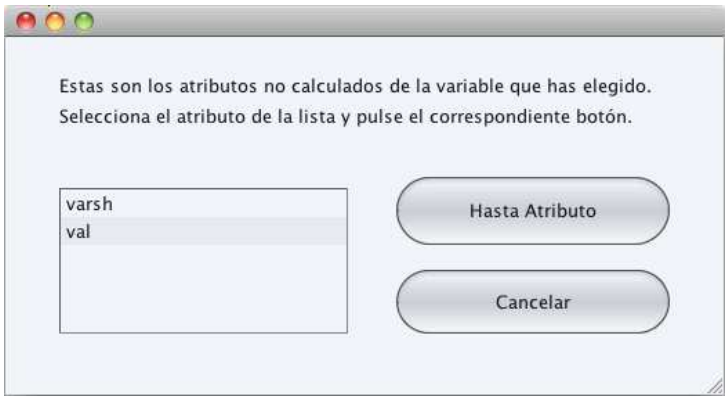


**Figura 3.3.3.** Pantalla destinada a la selección del atributo a calcular

**3)** Seleccionamos el atributo y pulsamos el botón correspondiente o cancelamos si no queremos llevar a cabo esta acción.

**4)** En ese momento comenzará la ejecución. En caso de haber seleccionado la animación, veremos como el árbol va creciendo poco a poco. En caso contrario, obtendremos directamente el nuevo árbol.

### *3.3.5 Idioma*

El depurador puede ser utilizado en inglés y castellano. Tenemos la opción de elegir el idioma en la pantalla donde seleccionamos la gramática y en la principal. En la primera de ellas, podemos hacerlo clickeando sobre una de las dos banderas que aparecen en su parte inferior izquierda (véase la Figura 3.2.1), mientras que en la principal debemos irnos al menú horizontal que encontramos en su parte superior. Por defecto, la herramienta está en castellano.

# 4. DESARROLLO E IMPLEMENTACIÓN

## 4.1 Introducción

El entorno de depuración Adebug se compone internamente de una serie de módulos que se encargan de procesar una serie de comandos con el objetivo de simular cómo funcionan los analizadores sintácticos por desplazamiento-reducción. Estos comandos seguirán un esquema que el depurador es capaz de interpretar obteniendo como salida el árbol de análisis sintáctico y el grafo de dependencias gracias a los cuáles sabremos cómo se van reduciendo las producciones, cómo se establecen las dependencias y en qué orden los atributos son evaluados.

En este capítulo se va a describir el desarrollo del sistema Adebug. En primer lugar encontramos información sobre el método que se ha llevado a cabo para el desarrollo de la aplicación. Posteriormente, se describe la estructura interna del depurador, cómo se relacionan cada uno de los módulos que lo componen para conseguir el objetivo final y, por último, algunos detalles de implementación.

## 4.2 Método de desarrollo

Para llevar a cabo el desarrollo del depurador se ha seguido una metodología incremental, basada en pequeñas iteraciones, dónde se presentaba lo realizado hasta el momento y se fijaban los próximos objetivos a seguir. Esto se ha realizado mediante una serie de reuniones con el director, donde se comentaba cuál era el estado del depurador en cada momento y cómo continuar trabajando en él.

En un principio se implementó la estructura interna que formaba la base del depurador, a partir de la cual se fueron añadiendo módulos. Una vez construida la estructura general, se estudió la forma en la que iba a ser representa para una correcta visualización que fuera fácil de entender y sencilla de manejar. Gracias a Piccolo y una serie de algoritmos de dibujo se consiguió obtener un buen resultado.

Hay que destacar, que cómo se ha comentado antes, la parte visual fue creada con posterioridad a la estructura interna. Es decir, independientemente de la interfaz, contamos con una estructura que es capaz de procesar una información de entrada (los comandos) y obtener el mismo resultado pero sin representación a nivel gráfico.

## 4.3 Arquitectura del sistema

### 4.3.1 Estructura general

La estructura interna del depurador está compuesta por cuatro paquetes contenedores de clases Java. A continuación se muestra la relación entre ellos:

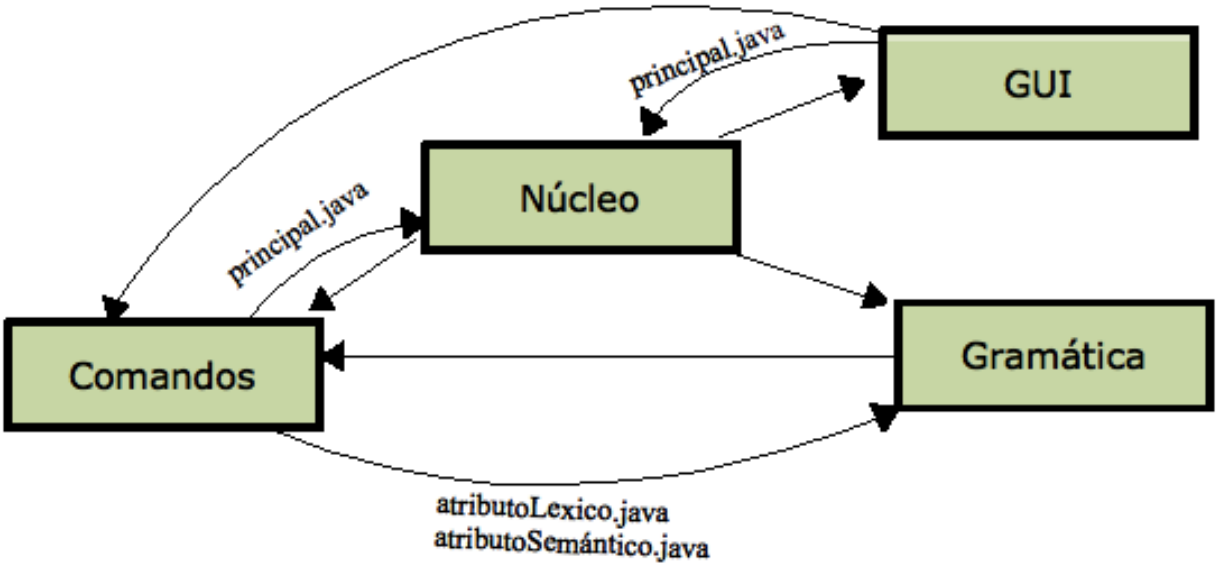


**Figura 4.3.1** Relación entre los paquetes que componen la estructura interna del depurador.

El paquete **GUI** hace uso de la clase principal.java. Si no quisiéramos una visualización gráfica, el paquete GUI podría ser casi totalmente eliminado.

El paquete **Comandos** utiliza las clases principal.java para ejecutar cada uno de los comandos. Las clases de los atributos léxicos y semánticos se utilizan ya que definen el tipo de los parámetros que utilizan dichos comandos.

El paquete **Gramática** tiene que importar los comandos, ya que es el que suministra la traza que los contiene.

El paquete **Núcleo** importa el resto de paquetes. Éste es el que llevará a cabo la ejecución y el almacenamiento de la estructura sobre la que se basa todo el procesamiento de la aplicación.

### 4.3.2 Módulos de Adebug

#### 4.3.2.1 Paquete Comandos

Este paquete está formado por la interfaz Comando y el resto de clases que la implementan. Estas clases representan cada uno de los comandos que nuestro depurador es capaz de interpretar con sus correspondientes accesoras y métodos auxiliares para su manejo.


**Figura 4.3.2** Diagrama UML de las clases del paquete Comandos

A continuación se explica la estructura de cada uno de los comandos que acepta la herramienta de depuración:

- **Reduce:**

Simboliza la reducción de una regla de producción y el establecimiento de los atributos de la cabecera de la producción. Tiene la siguiente estructura:

*Reduce (ident, numHijos, AtributosHeredados, AtributosSintetizados)*

La variable *iden* representa la etiqueta asociada a la parte derecha de la producción. La variable *numHijos* representa el número de nodos que encontramos en la parte izquierda de la producción. Por último, los atributos de su cabeza vienen representados en la linea superior mediante *AtributosHeredados* y *AtributosSintetizados.* Estos atributos son instancias de la clase *AtributoSemantico*.

- **Shift:**

Representa la acción de desplazamiento. Leerá el siguiente token de la entrada y se añadirá a nuestro árbol. Su estructura es la siguiente:

*Shift (ident, AtributosLéxicos)*

La variable *ident* representa la etiqueta asociada al nodo terminal que acaba de ser añadido y *AtributosLéxicos* representa la variable asociada con sus atributos. Estos atributos son instancias de la clase *AtributoLexico*.

- **Valor:**

Este comando será añadido a la traza cuando se conozca el valor de alguno de los atributos asociados a nodos cuyo valor era desconocido hasta el momento.

*Valor (IdentificadorAtributo, ValorResultado)*

Aquí la primera variable es un identificador numérico único asociado a cada uno de los atributos de los nodos y la segunda su nuevo valor que acaba de ser calculado.

- **Dep:**

Este comando representa el establecimiento de las dependencias entre atributos de dos nodos.

*Dep (AtributosRequeridos, AtributosDependientes)*

Aquí ambas variables simbolizan los identificadores numéricos asociados con los nodos.

#### *4.3.2.2 Paquete GUI*

En este paquete aparecen todas las clases java relacionadas con la visualización gráfica de los resultados del procesamiento del árbol. La pantalla de

selección de la gramática (*PantallaSeleccionGramatica.java*) es la primera que nos aparecerá al ejecutar el depurador. A continuación pasaremos a visualizar la pantalla principal (*Pantalla.java*), la cual contendrá un *JInternalFrame* donde se visualizará el grafo resultante (*PantallaGrafo.java*). Por último, otra de las pantallas que forman parte de la aplicación (*PantallaSeleccionVariable.java*) es la que aparece al clickear sobre los nodos del árbol con el objetivo de activar uno de los modos de ejecución de los que dispone nuestra herramienta de depuración (hasta la evaluación de un atributo). La clase *Grafo.java* es la que contiene los algoritmos necesarios para dibujar correctamente los árboles parciales y las dependencias. Es la clase que hace uso de Piccolo y que extiende a PCanvas que sirve para incorporar Piccolo a una aplicación de Java Swing. La clase *Posicion.java* tampoco tendría sentido que existiera sin la visualización gráfica. Cada instancia de esta clase representará la posición de alguno de los elementos que forman parte del grafo de dependencias mostrado. Por último, la clase *Dependencias.java* que ha sido creada con el objetivo de separar las dependencias para una clara visualización en el grafo. Un sólo comando puede llevar asociado dos flechas a ser representadas y éstas aparecerán en el grafo de forma síncrona una detrás de otra. Por ejemplo, el comando dep({5,6} , {7}) creará dos elementos de tipo Dependencia separando los dos identificadores que forman parte de los atributos requeridos ((5,7),(6,7)). Igualmente si aparece más de un atributo dependiente. El diagrama de clases de la figura 4.3.3 resume el diseño de este paquete.

#### *4.3.2.3 Núcleo*

Este paquete contiene, como su nombre indica, el núcleo de ejecución de la aplicación. Tenemos una serie de clases que constituyen la estructura interna de almacenamiento para un correcto procesamiento de comandos. Estas clases son las siguientes:

- **Arbol.java**

Simula la estructura del árbol que posteriormente se muestra en el canvas. En realidad es una lista de nodos que poseen a su vez un atributo de tipo árbol.

- **Nodo.java**

Representa cada una de los elementos que forman parte del árbol. Cada nodo cuenta por tanto con una serie de hijos, sus atributos heredados, sintacticos y léxicos.

- **ElementoPila.java**

Esta clase se utiliza sólo en el contexto del retroceso de acciones. Las instancias de esta clase son las que se añadirán a la pila para poder deshacer las distintas ejecuciones diferenciando, claro está, el tipo de ejecución que ha sido llevada a cabo.

- **tablaAtributosValor.java**

Es la lista que relaciona cada uno de los atributos asociados a nodos (sus identificadores numéricos con el valor que poseen en cada momento). En esta lista

aparecerán todos los atributos de nodos procesados y añadidos al árbol, independientemente de si su valor ha sido calculado o no.

• **tablaAtributosNombre.java**

Es la lista que relaciona, igual que en el caso anterior, los identificadores numéricos asociados a atributos con el nombre asociado que se le ha dado, bien sea en un comando de tipo Shift o de tipo Reduce.

•**tablaDependencias.java**

Es la tabla que se encargará de mantener almacenado el historial de las dependencias insertadas en el sistema mediante el comando Dep.

•**Dependencias.java**

Esta clase representa el tipo de los elementos que componen la lista anterior. En realidad está compuesta por dos listas: una con los atributos dependientes y la otra con los requeridos.

Además de estas clases, tenemos la clase **Principal.java** sobre la cual se centra todo el sistema de esta herramienta de depuración. En términos generales, esta clase contiene lo siguiente:

- Los métodos asociados a cada modo de ejecución hacia delante.
- Los métodos asociados a los distintos modos de ejecución hacia atrás.
- Los métodos de ejecución de cada tipo de comando.
- Métodos asociados al manejo de los Timers.
- Métodos auxiliares.

El diagrama de clases de la figura 4.3.4 resume el diseño de este paquete.

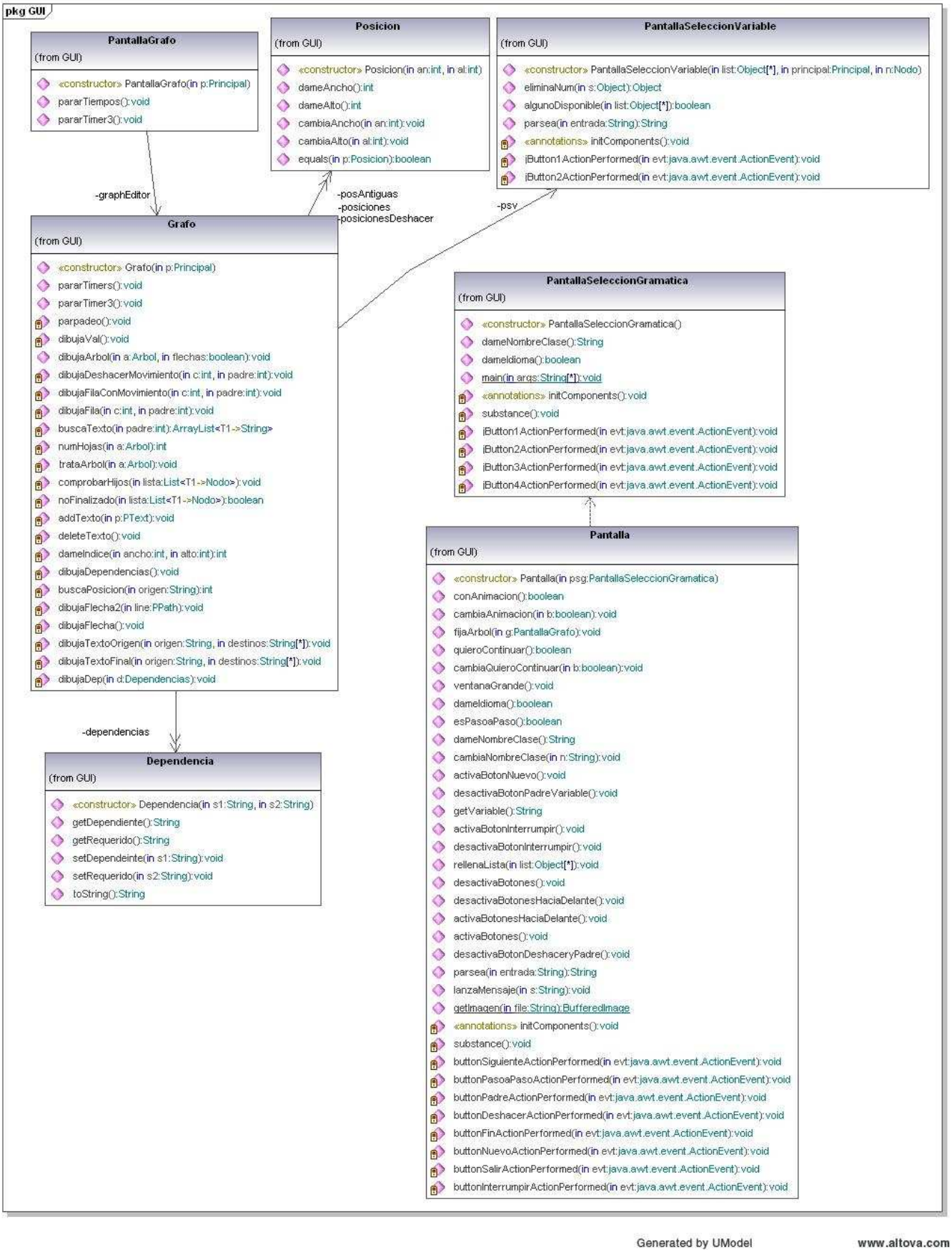


**Figura 4.3.3** Diagrama de clases del paquete GUI

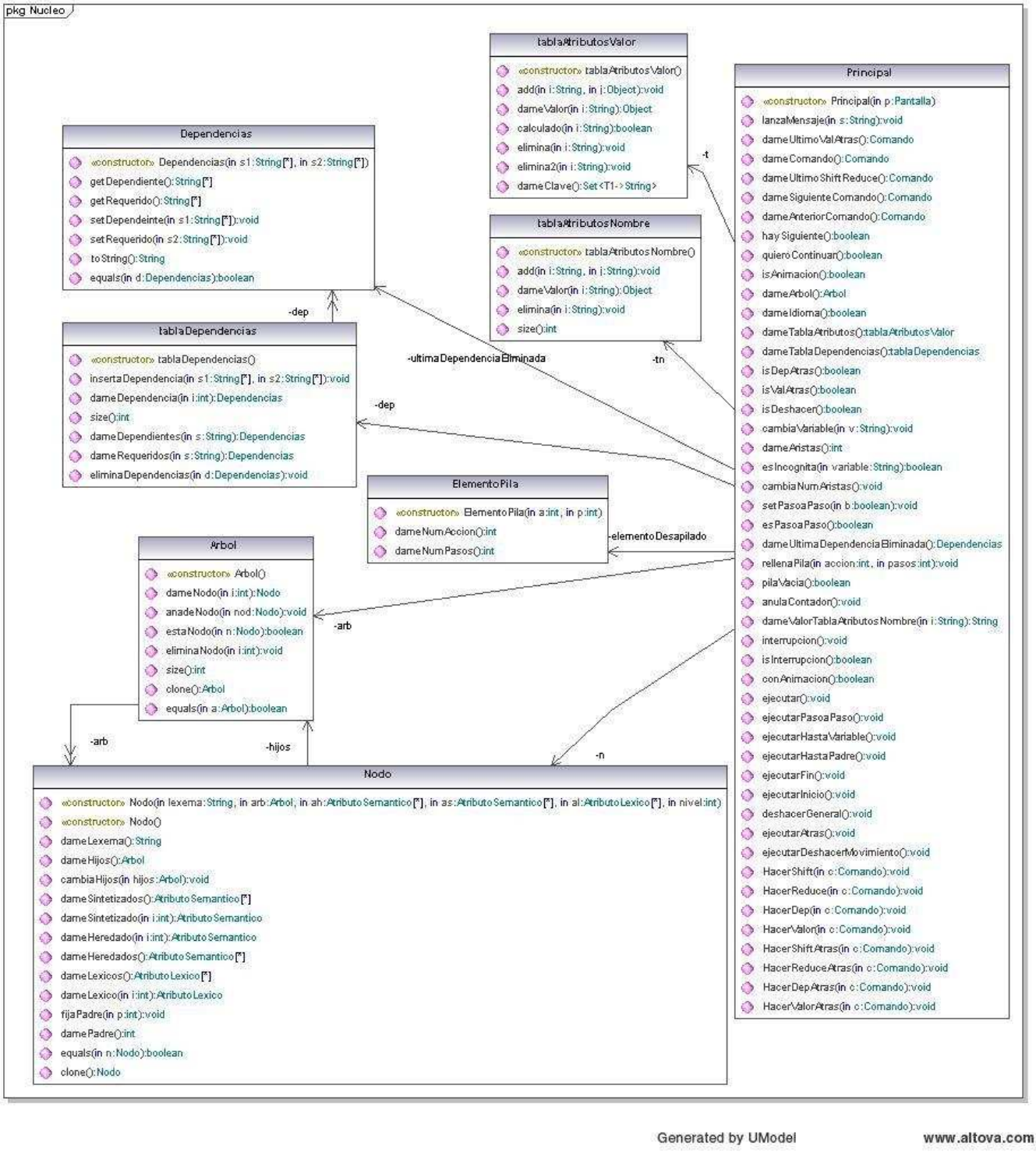


**Figura 4.3.4** Diagrama de clases del paquete Núcleo

#### *4.3.2.4 Gramática*

Este paquete introduce los siguientes componentes (véase Figura 4.3.6):

- **Interfaz Maquina**

Caracteriza el contrato que deben cumplir las fuentes de comandos aceptadas por ADebug.

- **Clase MaquinaI**

Una implementación de Maquina. Esta clase se instancia con el nombre de la clase que implementa la gramática de atributos instrumentada, y utiliza reflexión (método forName de la clase Class en Java) para cargar dinámicamente dicha gramática. Véase Figura 4.3.5.

```
public MaquinaI(String nombreGramatica) throws Exception {
    traza = new LinkedList<Comando>();
    Class laClase = Class.forName("Gramatica." + nombreGramatica);
    Gramatica g = (Gramatica) laClase.newInstance();
    traza = g.devolverTraza();
}
```

**Figura 4.3.5** Código asociado a la constructora de la clase MaquinaI

- **Interfaz Gramatica**

Esta interfaz caracteriza el contrato que deben cumplir las gramáticas de atributos instrumentadas cargadas dinámicamente por MaquinaI: deben proporcionar un método devolverTraza que devuelve la lista de comandos que entiende Adebug (Véase Figura 4.3.6)

Aparte de estos componentes, el paquete incluye también algunas clases de utilidad que pueden ser utilizadas para codificar gramáticas de atributos en herramientas de generación de traductores. Dichas clases se analizarán en el próximo capítulo.

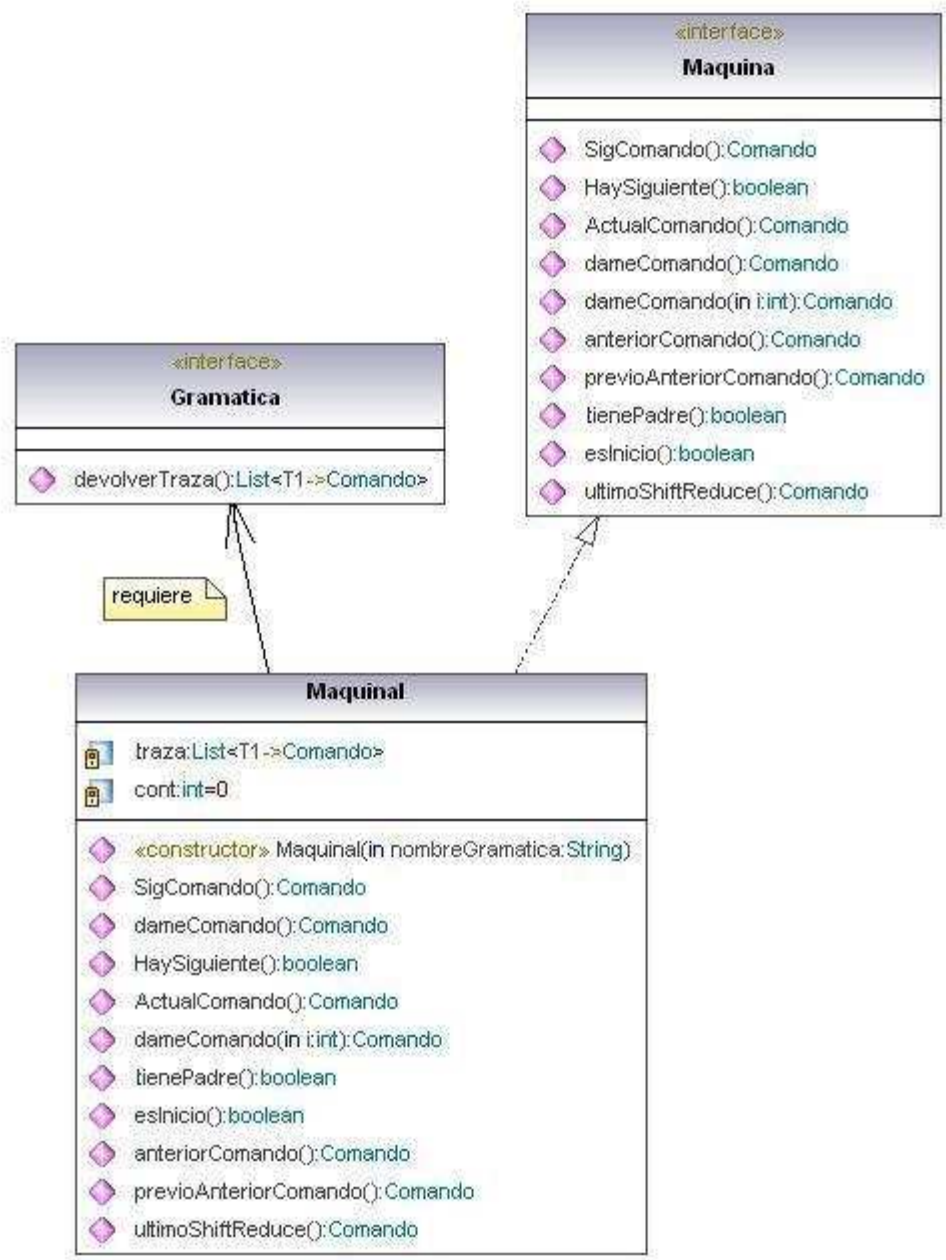


**Figura 4.3.6** Relación de clases del paquete gramática

### *4.3.3 Detalles de implementación*

A continuación se va a explicar, en términos de modelo interno, cómo funcionan tanto los diferentes comandos que el depurador analiza, como los diferentes modos de ejecución. Se explican también las estructuras necesarias para obtener un correcto

funcionamiento. Esto estará explicado de forma totalmente ajena a la interfaz que hemos utilizado para mostrar los resultados.

#### *4.3.3.1 Estructuras de almacenamiento*

Como se ha comentado anteriormente, internamente el depurador cuenta con una estructura que simula un árbol de análisis sintactico atribuido. Cada una de las acciones influirán de cierto modo en él. Dentro de este apartado de la memoria se explicarán las estructuras internas que componen el depurador y de qué forma el procesamiento de cada acción influye en el árbol y en otras estructuras internas que lo apoyan para la correcta interpretación de los comandos.

##### *4.3.3.1.1 Árbol interno*

La estructura de árbol que se ha mencionado previamente no es más que una lista de elementos de tipo nodo, de tal modo que esta lista tendrá tantos elementos como nodos se encuentre en el nivel 0 del árbol (por tanto, más que un árbol es, en realidad, un bosque; no obstante, mantendremos la denominación de árbol, ya que, al final, es la estructura a la que converge). Es decir, si tenemos un árbol parcial como el mostrado en la Figura 4.3.7, el número de elementos de la lista será 3.

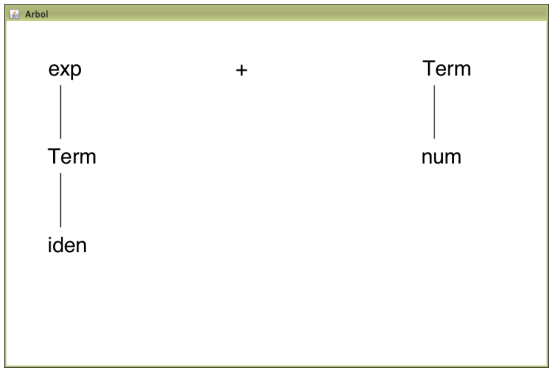


**Figura 4.3.7** Imagen del desarrollo de la ejecución en un momento determinado.

La estructura del árbol es recursiva, cada uno de los nodos que componen la lista, entre otros atributos, tienen a su vez un atributo de tipo árbol (Figura 4.3.8).

```
private String lexema;
private Arbol hijos;
private AtributoSemantico[] ah;
private AtributoLexico[] al;
private AtributoSemantico[] as;
private int PosicionPadre;
```

**Figura 4.3.8** Atributos de la clase nodo.

Como podemos ver en la figura superior, cada nodo cuenta con una serie de elementos que nos sirven para representar los atributos heredados, sintetizados y léxicos de cada nodo terminal o no terminal de la gramática. El lexema es la etiqueta que representa a cada nodo.

Por último, se puede ver que, en cada nodo, tenemos representada la posición del padre dentro de la lista de nodos en la que se encuentra. En caso de no tener asociado ningún padre, por encontrarse en el nivel 0 del árbol, esta variable tendrá un valor de -1. Esta variable estará únicamente destinada para el dibujo del árbol en la interfaz.

#### *4.3.3.1.2 Almacenamiento de dependencias.*

Las dependencias establecidas por el comando 'dep' deben ser almacenadas internamente. De este modo, cuando se conozca el valor del atributo requerido, el de los atributos dependientes también será conocido inmediatamente después. Esta estructura tiene que ser mantenida, y es especialmente importante para la visualización del árbol en la interfaz, ya que las dependencias entre atributos son mostradas cuando el valor del atributo dependiente ha sido calculado. Este almacenamiento necesario será implementado mediante una lista (Figura 4.3.9), dónde cada elemento representará cada uno de los comandos 'dep' que han sido ya procesados.

```
private List<Dependencias> dep;
```

**Figura 4.3.9** Almacenamiento de las dependencias

Cada una de estas dependencias está formada por dos listas. La primera de ellas mantendrá los atributos requeridos y la segunda los dependientes (en ambos casos los identificadores numéricos asociados a ellos).

#### *4.3.3.1.3 Almacenamiento de valores de atributos*

A medida que el procesamiento de los comandos se va llevando a cabo, necesitamos también alguna forma de saber los atributos que han sido ya calculados y cuáles no. Internamente, estos atributos son identificados numéricamente de forma unívoca y para poder trabajar con ellos tenemos que relacionar estos números con el valor asociado y con el nombre del atributo al que representan. Para ello, contamos con dos clases:

- *tablaAtributosNombre:* Relaciona el identificador con el nombre asociado a dicho atributo.
- *tablaAtributosValor:* Relaciona el identificador con el valor asociado. Cuando este atributo no haya sido calculado, el valor que tendrá será "???".

Ambas clases contienen un *HashMap* que nos permitirá fácilmente acceder al valor o a la etiqueta, dado el identificador.

### 4.3.3.2 Modos de ejecución hacia delante

#### 4.3.3.2.1 Ejecución siguiente

Este tipo de acción, ejecutará todos los comandos necesarios hasta encontrar el primer comando Shift o Reduce y sus posibles posteriores dependencias o evaluaciones.

#### 4.3.3.2.2 Ejecución Paso a Paso

Este tipo de ejecución es el núcleo principal de todas las demás, ya que todas ellas recurrirán a la ejecución paso a paso tantas veces como sea requerido. Se ejecutarán comandos hasta encontrar uno que no sea una dependencia. Si una dependencia es encontrada, se llevará a cabo su acción correspondiente de forma interna, pero seguiremos avanzando y ejecutando un nuevo comando hasta encontrar uno que sea del tipo Reduce, Shift o Valor. Podemos ver en la Figura 4.3.10. el núcleo principal de procesamiento, dónde *m* es una variable de la clase *máquina,* que será la que se encargará de suministrar la traza de comandos.

```
if (m.HaySiguiente()) {
    c = m.SigComando();
    if (c instanceof Shift) {
        ((Shift) c).ejecutar(this);
    } else if (c instanceof Reduce) {
        ((Reduce) c).ejecutar(this);
    } else if (c instanceof Dep) {
        ((Dep) c).ejecutar(this);
    } else if (c instanceof Valor) {
        ((Valor) c).ejecutar(this);
    }
```

**Figura 4.3.10** Parte del código del método asociado a la ejecución "paso a paso".

#### 4.3.3.2.3 Ejecución hasta el nodo padre

Los comandos serán ejecutados hasta encontrar un comando reduce que asocie un nodo no terminal con sus hijos, siendo uno de estos hijos el último nodo que ha sido añadido a la estructura interna del árbol. También ejecutará las comandos de tipo 'dep' que se encuentren inmediatamente después de dicho comando 'Reduce'. Si la ejecución no ha comenzado, este tipo de ejecución no podrá llevarse a cabo, ya que no tendremos ningún nodo del que partir en la búsqueda del nodo padre.

#### 4.3.3.2.4 Ejecución hasta el cálculo de un atributo

La ejecución de comandos en este tipo de ejecución finalizará cuando el valor de un cierto atributo sea conocido (véase la Figura 4.3.11); es decir, cuando sea ejecutado un comando 'Valor' cuyo identificador corresponda con el del atributo que nosotros hemos seleccionado.

```
ejecutarPasoaPaso();
contadorVariable++;
Comando c = m.anteriorComando();
if (!(c instanceof Valor) || !(((Valor) c).getAtributo().equals(variable)))
        ejecutarHastaVariable();
```

**Figura 4.3.11** Fragmento representativo del código del método asociado a este tipo de ejecución.

#### *4.3.3.2.5 Ejecución hasta el final*

Como su nombre indica, la ejecución de comando no parará hasta llegar al último comando de la traza. Véase Figura 4.3.12.

```
while (m.HaySiguiente()) {
    contadorFin++;
    ejecutarPasoaPaso();
}
```

**Figura 4.3.12** Fragmento representativo del método asociado a la ejecución hasta el final.

La variable contadorFin tiene como única finalidad favorecer el retroceso de la acción (véase siguiente apartado).

### *4.3.3.3 Modos de ejecución hacia detrás*

#### *4.3.3.3.1 Retroceso de la última acción*

Para poder realizar el retroceso de acciones, internamente se ha utilizado una pila. Esta pila almacenará elementos de *tipoPila* que estarán constituidos por un identificador del tipo de acción que se ha llevado a cabo y el número de pasos que han sido realizados. Todos las acciones de retroceso llaman a un método llamado ejecutarAtrás cuyo núcleo principal se basa en el código mostrado en la Figura 4.3.13.

```
if (c instanceof Shift) {
    ((Shift) c).ejecutarAtras(this);
} else if (c instanceof Reduce) {
    ((Reduce) c).ejecutarAtras(this);
} else if (c instanceof Dep) {
    depAtras = true;
    ((Dep) c).ejecutarAtras(this);
} else if (c instanceof Valor) {
    valAtras = true;
    ((Valor) c).ejecutarAtras(this);
}
```

**Figura 4.3.13** Fragmento del código del método ejecutarAtrás.

Según el comando tratado se llamará al correspondiente método de retroceso asociado al tipo de comando. Cada uno de estos métodos utilizará la variable numPasos añadida a la pila por cada acción activada. Se llamará, por tanto, tantas veces al método ejecutarAtrás como indique dicha variable.

#### *4.3.3.3.2 Retroceso hasta el inicio*

Esto se lleva a cabo mediante el vaciamiento completo de la pila, volviendo por tanto, al estado inicial. Tanto la pila, el árbol y las listas utilizadas para almacenamiento se encontrarán completamente vacías.

#### *4.3.3.4 Visión interna de los comandos*

A continuación se explica cómo actúa internamente los diferentes comandos, tanto en sus métodos de ejecución hacia delante como en los de retroceso.

**Reduce**

Cuando el comando Reduce es ejecutado internamente, se realizan los siguientes cambios:

- Creación del nodo asociado a la parte izquierda de la producción de la gramática que ha generado el comando.
- Los nodos de la parte derecha de la producción pasaran a un nivel 1, mientras que el nodo padre (parte izquierda de dicha producción), pasará a formar parte del nivel 0.
- Incorporación de los atributos heredados y sintetizados a las listas que almacenan su identificador numérico, su valor y su nombre.

El método que lleva a cabo el retroceso de este tipo de comandos internamente realiza lo siguiente:

- Elimina el nodo padre y sus sucesores suben de nivel. De esta forma, sus hijos pasarán a estar en el nivel dónde estaba el padre (nivel 0).
- Los atributos son eliminados de las tablas comentadas anteriormente. Esto se hace fácilmente porque el identificador numérico actúa como clave de las tablas Hash.

**Shift**

La acción interna de este comando es similar a la de un comando Reduce, pero algo más simple ya que no tiene hijos asociados a él. En cuanto a su ejecución hacia delante tenemos lo siguiente:

- Un nuevo nodo es creado y añadido al árbol en la lista de nodos del nivel 0.
- Sus atributos léxicos son añadidos a las tablas del mismo modo que se hace con los atributos sintetizados y heredados del comando Reduce.

En el retroceso se lleva a cabo la:

- Eliminación del último nodo de la lista del árbol; es decir, el último nodo añadido en el nivel 0.
- Eliminación de los atributos léxicos de las correspondientes listas.

**Valor**

La ejecución de este comando tiene internamente la misión de añadir a la lista de identificadores y valores el nuevo valor que se acaba de obtener. Esto será facilmente logrado dado que el comando Valor nos suministra los dos parámetros (identificador de atributo y valor) que serán los usados para ello. En caso de querer deshacer la ejecución del comando bastará con eliminar de la lista el valor asociado al identificador de atributo dado. Con el objetivo de visualizar el grafo en la interfaz, el último comando Valor que ha sido desecho será guardado internamente.

**Dependencia**

Para ejecutar y deshacer un comando 'dep' será suficiente con añadir la correspondiente entrada a la lista de dependencias (con los dos identificadores numéricos que forman parte del comando) o eliminarlo, si así se requiere. Al igual que se ha comentado con el comando Valor, la última dependencia que ha sido deshecha también se guardará para favorecer su representación visual.

#### *4.3.3.5 Clase Timer*

Con el fin de animar el crecimiento del árbol, se necesita una herramienta que controle el tiempo en el que cada movimiento es realizado. Aquí surge la utilización de la clase Timer de java. Esta clase dispara un evento después de un retardo especificado y cuenta con una serie de métodos que nos permiten fácilmente controlar estos intervalos de tiempo. El depurador hace uso de esta clase en los siguientes casos:

- Para añadir nodos al árbol. Este timer será el más externo, y será llamado dentro de los distintos modos de ejecución en la clase Principal.java. El resto de los timers que aparezcan serán ejecutados de forma interna a este último, que en ese momento está parado esperando una reanudación.

- Cuando tenemos un atributo cuyo valor acaba de ser calculado, se mostrará en primer lugar el nombre de los atributos requeridos para ese cálculo, después las flechas correspondientes y por último el nombre del atributo recién calculado. Para secuenciar esto, necesitamos también usar la clase Timer.

- Parpadeo de las flechas

- Parpadeo del texto mostrado en la evaluación de un atributo.

Una vez que un timer es activado, ejecutará una rutina cada cierto tiempo. Ésta será ejecutada hasta que deje de cumplirse alguna condición necesaria, en cuyo caso se procede a parar el timer.

#### *4.3.3.6 Dibujado del árbol*

A continuación se encuentra una explicación de los pasos que se siguen internamente para dibujar el árbol de análisis atribuido de una forma apropiada. La

estructura de árbol que teníamos (ver apartado 3.3.1.1), pasa a transformarse a una lista de nodos, donde cada nodo tiene conocimiento de en qué parte de dicha lista se encuentra su nodo padre. Manteniendo esta estructura es mucho más fácil acceder a cada nodo según dibujamos el árbol. Para ello, se empezará a dibujar desde su primer nivel, es decir, aquellos nodos que no tienen padre, bien porque todavía no se ha reducido la correspondiente regla o porque es la cima del árbol y por tanto el nodo correspondiente con la parte derecha de la producción más externa. Los nodos del primer nivel, se dibujarán en el PCanvas de forma uniforme, es decir, el espacio se distribuirá en función de cuántos nodos hay. A partir de ahí, cuando se quiera dibujar un nodo que no tiene hermanos, se partirá de la posición del padre y se le añadirá sólo altura sin variar su ancho. En cambio, cuando haya que dibujar un nodo con un hermano o más, se distribuirá el espacio entre el número de ellos. Es decir, dada la posición en anchura del padre, se asociará un rango de espacio a su derecha y a su izquierda, de forma que la colocación en anchura de los hijos se distribuya entre dicho espacio de forma uniforme pero a una altura inferior.

Las posiciones que les damos a los nodos, son almacenadas en una estructura en el mismo orden en el que aparecen en la lista de nodos creada anteriormente. Esto nos permite acceder fácilmente a la posición de un determinado nodo con posterioridad a ser dibujado. Esto es necesario cuando mostramos las flechas indicadoras de dependencia o cuando aparece el texto asociado a la evaluación de los atributos. Igualmente necesario es mantener esta estructura para un correcto funcionamiento del modo de ejecución de avance hasta la evaluación de un atributo, ya que al pinchar sobre uno de los nodos, Piccolo nos da la posición dónde hemos clickeado y a partir de ahí nosotros tenemos que saber qué nodo de los almacenados es el que tiene el atributo deseado.

# 5. CODIFICACIÓN DE GRAMÁTICAS CON BYACC Y DEPURACIÓN CON ADEBUG.

## 5.1 Introducción

YACC (Yet Another Compiler-Compiler)[5] es un generador de analizadores sintácticos a partir de una gramática LALR(1). Yacc recibe cómo entrada una secuencia de los componentes elementales de un texto, y comprueba si esa secuencia se ajusta a la estructura definida por dicha gramática. Los componentes elementales recibidos son sus símbolos terminales. Esta gramática tiene que ser especificada de acuerdo a unas reglas que deben ser seguidas para obtener un correcto procesamiento. Yacc generará el código para ese analizador sintáctico. Lo que nosotros queremos es que dicho analizador codifique una gramática de atributos, que esté escrito en java, y que esté adecuadamente instrumentado, para poder realizar la depuración con Adebug. Para ello utilizaremos la versión **Byacc/J**. Byacc (Berkeley Yacc) es una de las variantes de Yacc escrita por Robert Corbett en 1990[6].

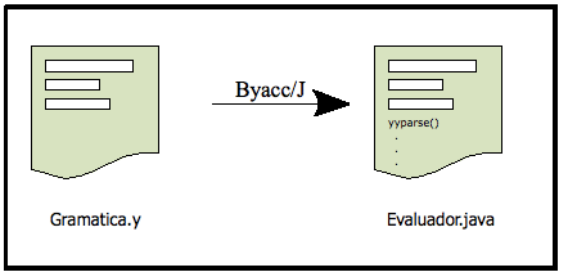


**Figura 5.1.1** Obtención de un analizador sintático escrito en java a partir la especificación Yacc de una gramática.

[5] Johnson, S.C. YACC: Yet Another Compiler-Compiler. Unix Programmer's Mannual Vol 2b, 1979.

[6] Hurka T.BYacc's Manual. Disponible en http://byaccj.sourceforge.net/. 2008

El archivo resultante tendrá por defecto el nombre de *Parser.java.* Este nombre puede ser modificado mediante el comando *-JClass = <nombre de la clase>* y será el que se tendrá que escribir cuando el depurador lo requiera al principio de la ejecución de la aplicación.

Una especificación en BYacc/J está compuesta por tres secciones que deben ser claramente diferenciadas (véase Figura 5.1.2):

- **Cabecera:**

Las declaraciones de importación y las relacionadas con los paquetes java deben estar escritas aquí. Este código se trasladará literalmente al fichero de salida, al principio del código generado (debido a que está limitado por %{ y %}). Las definiciones de los tokens, tipos de datos y precedencia de la gramática también tendrán que ser escritas en esta sección.

Tienen que cumplir los siguientes formatos:

- **Definición de tipos**

*%type tipo exp*

Se indica qué tipo tienen los símbolos no terminales.

- **Definición de tokens**

Cada definición tiene la siguiente estructura:

*%token nombreToken1 [RestoTokens]*

Se usa para definir los tokens de la gramática. Representarán cada uno de sus símbolos terminales.

- **Definición de precendencias**

Bajo el siguiente formato:

*%[left|right] nombreOperador*

Sirve para indicar si se quiere que el operador agrupe a derechas o a izquierdas. La definición de precedencias es importante para la desambiguación del lenguaje.

Pueden también aparecer otras líneas de declaración **%union**, **%start** y**%nonassoc**.

- **Acciones:**

Aquí se encuentra la definición de las acciones asociadas a las reglas de la gramática. Las producciones de la gramática que tenían esta forma:

*<lado izquierdo> ::= <alt 1> | <alt 2> | ... | <alt n>*

pasan a escribirse de la siguiente forma:

*<lado izquierdo> : <alt 1> {acción semántica 1}*

*| <alt 2> {acción semántica 2}*

*.*

*.*

*.*

*.*

*| <alt n> {acción semántica n}*

*}*

Estas acciones semánticas están formadas por una serie de proposiciones donde $$ se refiere al valor del registro semántico asociado al no terminal de la cabecera de la producción, mientras que $i se refiere al registro semántico asociado con el i-ésimo símbolo gramatical del lado derecho de la producción. Estas proposiciones son ejecutadas siempre que se reduzca por la producción asociada. La acción semántica por defecto es *{$$=$1;}.*

- **Código:**

Aparecerá aquí la implementación de algunas funciones que pueden ser útiles, bien por que van a ser llamadas desde nuestro código o por las reglas semánticas asociadas a las producciones. Las funciones más comunes que suelen aparecer son main() e yyerror(). La primera de ellas, se utiliza para personalizar el programa mientras que la segunda, se utiliza para indicar el comportamiento deseado en caso de encontrar un error sintáctico.

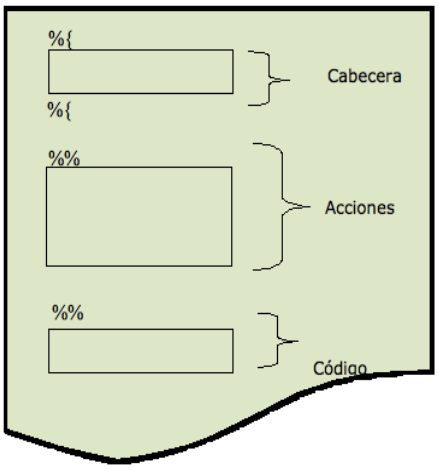


**Figura 5.1.2** Muestra la correcta estructura que debe tener el archivo Yacc. Los simbolos externos a las secciones deben de ser incorporados.

## 5.2 Codificación de gramáticas de atributos

Para facilitar la codificación de gramáticas de atributos con BYacc se hace uso de una serie de clases Java que están incluidas en el paquete *Gramatica* de la herramienta Adebug. A continuación, se explica con detalle cada una de estas clases que sirven de apoyo a la codificación y en el siguiente apartado se encuentra un ejemplo de codificación de una gramática con BYacc. La Figura 5.2.1 esquematiza dichas clases.

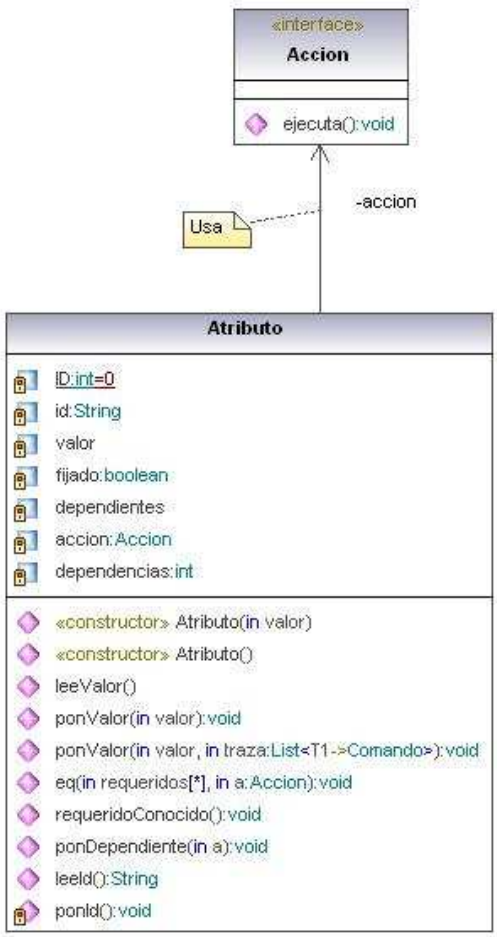


**Figura 5.2.1** Relación entre las clases *Accion* y *Atributo*.

- **Accion.java**

Esta clase es una interfaz que consta de un único método *ejecuta*(), el cual es implementado cada vez que se crea una instancia de tipo Accion (véase Figura 5.2.1). Los objetos de tipo acción encapsularán la implementación de las ecuaciones semánticas de la gramática.

```
public interface Accion {
  void ejecuta();
}
```

**Figura 5.2.2** Clase acción

- **Atributo.java**

La principal característica de esta clase es que es la encargada de lanzar las ejecuciones de las acciones. Es decir, cada atributo tendrá asociados los atributos de los que depende. De tal forma que el valor de dicho atributo no se conocerá hasta conocer el valor de los demás. Una vez que estos sean conocidos, se llevará a cabo alguna tarea sobre ellos y se obtendrá el valor del atributo dependiente. Para realizar esto, se utiliza una estructura de pila donde iremos almacenando cada uno de los atributos de los que depende otro atributo previo. Según se vaya conociendo el valor de estos atributos, se van desapilando hasta que la pila quede vacía en cuyo momento se llevará a cabo la ejecución de la acción asociada al atributo dependiente.

```
valOfExp.eq(new Atributo[]{valOfTerm},
                new Accion() {
                  public void ejecuta() {
                    valOfExp.ponValor(valOfTerm.leeValor(),traza);
                  }});
```

**Figura 5.2.3** Ejemplo de instalación de una acción para calcular el valor de un atributo.

Como se puede ver en la Figura 5.2.2, se está creando una acción. Esta acción consiste en añadir a la traza un comando de tipo Valor. Ésto es lo que nos interesa desde el punto de vista de Adebug, pero la acción asociada a la evaluación de un atributo varía dependiendo del contexto en el que estemos trabajando. El fragmento que se ve en la figura, está asociado a la producción *Exp -> Term*, y la misión del método *eq* es la de apilar el atributo sintetizado de Term en la pila de elementos requeridos para calcular el valor del atributo sintetizado de Exp y la de asociar la evaluación de dichos valores con la acción que se llevará a cabo.

## *5.3 Ejemplo de codificación de una gramática con BYacc*

Para conseguir un correcto entendimiento de la codificación, se explicará mediante un ejemplo los pasos que tienen que realizarse.

Como ya hemos comentado anteriormente, el núcleo principal de un archivo *yacc* son un conjunto de reglas de traducción, cada una de las cuales cuenta con una

producción de la gramática y la acción semántica asociada. Los elementos que intervienen en cada producción tienen asociado por defecto un objeto. En nuestro caso, estos objetos serán instancias de la clase *Sem* y como se puede ver en la Figura 5.3.1, esta clase está compuesta por tres elementos destinados a la respresentación de los atributos sintetizados, heredados y léxicos de cada uno de los símbolos que componen la gramática. Dicha clase estará definada en la cabecera del programa yacc.

```
class Sem {
  Atributo<Integer> val;
  Atributo<Map<Character,Integer>> varsh;
  Atributo<Character> lex;
}
```

**Figura 5.3.1** Definición de la clase Sem. Como podemos ver cuenta con tres atributos necesarios para la correcta generación de los comandos deseados.

Dada la gramática que queremos depurar, siendo Exp la cabeza de la regla más externa (ver Figura 5.3.2), hay que añadirle un axioma:

*S ->Exp (S : Exp en Byacc)*

Esto se hace porque nos interesa llevar a cabo una serie de acciones para favorecer el procesamiento. En este caso, se creará la estructura que contendrá los atributos que se irán heredando hacia los nodos de nuestro árbol (tabla de atributos heredados). En la Figura 5.3.3, podemos observar cómo ésto se realiza. Su última línea, en la cual se llama al método *ponValor,* está destinada a la generación de la traza de comandos que Adebug requiere. Por lo tanto, esta linea no formará parte de la codificación de la gramática, sino que está destinada al proceso de depuración.

```
Exp → Exp + Term   { Exp1.varsh = Exp0.varsh
                     Term.varsh = Exp0.varsh
                     Exp0.val = Exp1.val + Term.val }

Exp → Term         { Term.varsh = Exp. Varsh
                      Exp.val = Term.val }

Term → num         { Term.val = num.val }

Term → ident       { Term.val = valorDe(Term.varsh, ident.lex) }
```

**Figura 5.3.2** Gramática que va a ser traducida a Byacc.

```
Sent : Exp {
  Map<Character,Integer> vars = new HashMap<Character,Integer>();
  vars.put('x',10);
  varsh($1).ponValor(vars,traza);
}
```

**Figura 5.3.3** Producción de la regla *Sent -> Exp* transformada a Byacc

A cada una de las producciones de la gramática que ya han sido transformadas según la sintaxis de BYacc, tenemos que añadirles ciertas extensiones de código, para favorecer el manejo de los atributos asociados a los símbolos de la gramática. Esta extensión se llevará a cabo de forma similar en cada una de las producciones. Dada la gramática de la Figura 5.3.2, para codificar la primera producción:

*Exp -> Exp + Term*

Tenemos que añadir al código las siguientes extensiones:

1. Creación del objeto asociado a la cabeza de la producción.

   Siendo $$ el identificador para referirnos al registro semántico asociado con la cabeza, el código tendría la siguiente forma, donde:

   **$$.obj = new Sem();**

   **sem($$).varsh = new Atributo<Map<Character,Integer>>();**

   **sem($$).val = new Atributo<Integer>();**

   Creamos el objeto Sem y sus Atributos. Hay que fijarse que la clase Sem también tiene un atributo *lex,* que en ese caso, no nos interesa porque no se trata de la producción asociada con un terminal.

2. Creación de variables asociadas a los atributos de los nodos con el objetivo de poder trabajar con ellas y establecer así el orden en su evaluación y, por tanto, el lanzamiento de las acciones que estas evaluaciones generan.

   **final Atributo<Map<Character,Integer>> varshOfExp0 = varsh($$);**

   **final Atributo<Map<Character,Integer>> varshOfExp1 = varsh($1);**

   **final Atributo<Map<Character,Integer>> varshOfTerm = varsh($3);**

   **final Atributo<Integer> valOfExp0 = val($$);**

   **final Atributo<Integer> valOfExp1 = val($1);**

   **final Atributo<Integer> valOfTerm = val($3);**

3. Creación de las acciones asociadas con la evaluación de atributos. Se establecen las dependencias entre los símbolos de la gramática y las citadas acciones que

deben ser lanzadas. A continuación se muestran estas asociaciones para cada una de las ecuaciones de la regla de producción que estamos analizando.

$Exp_1$.varsh = $Exp_0$.varsh

```
varshOfExp1.eq(new Atributo[]{varshOfExp0},
  new Accion() {
        public void ejecuta() {
        varshOfExp1.ponValor(varshOfExp0.leeValor());
 }});
```

Term.varsh = $Exp_0$.varsh

```
varshOfTerm.eq(new Atributo[]{varshOfExp0},
   new Accion() {
        public void ejecuta() {
        varshOfTerm.ponValor(varshOfExp0.leeValor());
 }});
```

$Exp_0$.val = $Exp_1$.val + Term.val

```
valOfExp0.eq(new Atributo[]{valOfExp1,valOfTerm},
new Accion() {
      public void ejecuta() {
      valOfExp0.ponValor(valOfExp1.leeValor()+valOfTerm.leeValor());
 }});
```

Como se puede ver, se ha hecho uso de un método **eq(atributo,acción)**, el cual establece la relación entre el cálculo de un atributo y la acción que conlleva el conocimiento de este nuevo valor. Acción es una interfaz cuyo método a ser implementado es *ejecuta().* De este modo, cuando el valor de dicho atributo se conoce, se decrementa el número de atributos de los que depende. Si todos los atributos de los que depende ya han sido calculados, se ejecutará la acción que dicho atributo tiene asociada. En el tercer de los ejemplos escritos anteriomente podemos ver que el atributo *val* de $Exp_0$ depende de los atributos *val* de $Exp_1$ y Term. Cuando estos valores se calculen, se ejecutará la acción correspondiente.

Los símbolos terminales de la gramática *iden* y *num* tienen también atributos que deben ser tratados como tal y necesitamos poder manejarlos. Para ello debemos

modificar las producciones cuya parte derecha corresponde con un terminal obteniendo otras dos producciones.

**Term -> num** { **Term -> NUM**, **NUM -> num** }

**Term -> ident** { **Term -> IDENT**, **IDENT -> ident** }

Las producciones *NUM -> num* e *IDENT -> ident* se encargarán de crear el objeto asociado a los símbolos que eran terminales de la gramática inicial.

```
IDEN : iden {
$$.obj = new Sem();
sem($$).lex = lex($1);
}
```

```
NUM : num {
$$.obj = new Sem();
sem($$).val = val($1);
}
```

## 5.4 Depuracion con BYacc – Adebug

Una vez que la gramática deseada ha sido codificada, si se quiere que sea depurada con la herramienta Adebug, esta codificación tiene que ser extendida con el objetivo de suministar la traza de comandos que el depurador es capaz de interpretar.

En primer lugar tenemos que crear la estructura que mantendrá dicha lista y en la cual se irán añadiendo los correspondientes comandos según corresponda. Esta estructura recibirá el nombre de *traza* (véase Figura 5.4.1) y cuya definición aparecerá en el último apartado de un archivo Yacc (apartado destinado al código).

```
private List<Comando> traza = new LinkedList<Comando>();
```

**Figura 5.4.1** Creación de la lista de comandos.

Así mismo, la clase generada por BYacc debe implementar la interfaz Gramatica. Para ello:

- El archivo de especificación debe ser procesado con la opción -I Gramatica, que indica a BYacc que la clase generada implemente la interfaz *Gramatica*.
- Debe implementarse, así mismo, el método devolverTraza. Dicho método invocará el método parse (punto de entrada al traductor generado por BYacc), y devolverá el valor de traza.

### 5.4.1 Creación de los comandos

Como se ha dicho con anterioridad, los comandos que el depurador reconoce pueden venir de diferentes fuentes. Utilizamos Byacc como una forma de obtención de los comandos a partir de la interpretación de la correspondiente gramática. A continuación se recuerdan los tipos de comandos que el depurador interpreta y en qué partes del código del archivo BYacc deben ser añadidos a la traza.

#### 5.4.1.1 Comando Reduce

Simboliza la reducción de una regla de producción y el establecimiento de los atributos de la cabecera de la producción. Tiene la siguiente estructura:

*Reduce (Ident, NumHijos, AtributosHeredados, AtributosSintetizados)*

- *Ident* representa el identificador asociado a la cabeza de la regla de producción.
- *NumHijos* es el número de símbolos en la parte derecha de la producción.
- *AtributosHeredados* y *AtributosSintetizados* son las listas de los atributos heredados y sintetizados asoociados a la cabeza de la producción.

Este tipo de comandos deben ser escritos en cada una de las producciones de la gramática ya que simbolizan sus respectivas reducciones. A continuación encontramos algunos ejemplos de creación de comandos de tipo Reduce junto con las producciones a las que acompañan.

```
Exp –> Exp + Term

traza.add(new Reduce("Exp", 3,
        new AtributoSemantico[]{newAtributoSemantico(varsh($$).leeId(),"varsh")},
        new AtributoSemantico[]{new AtributoSemantico(val($$).leeId(),"val")}));
Exp –> Term
  traza.add(new Reduce("exp",1,
        new AtributoSemantico[]{new AtributoSemantico(varsh($$).leeId(),"varsh")},
         new AtributoSemantico[]{new AtributoSemantico(val($$).leeId(),"val")}));
```

Como se ha dicho anteriormente, && simboliza el registro semántico asociado a la cabeza de la producción. Por tanto, *varsh(&&).leeId()* y *val(&&).leeId()* son los identificadores asociados con el atributo heredado y sintetizado de la cabeza de la regla.

### 5.4.1.2 Comando Shift

Representa la acción de desplazamiento. Leerá el siguiente token de la entrada y se añadirá a nuestro árbol. Su estructura es la siguiente:

*Shift (Ident, AtributosLéxicos)*

- *Ident* es el identificador del símbolo que se mostrará en el árbol.
- *AtributosLéxicos* representa la lista de atributos léxicos asociados al símbolo terminal.

Estos comandos se añaden a la traza en las producciónes de simbolos terminales. Dada la gramática de la Figura 5.3.2 y su correspondiente transformación al lenguaje que BYacc interpreta, las siguientes acciones (escritas junto con sus correspondientes producciones transformadas) serán llevadas a cabo para añadir a la traza el comando Shift:

```
Num : num

traza.add(new Shift("num",

                    new AtributoLexico[]{new
                    AtributoLexico(val($1).leeId(),"num",val($1).leeValor())}));

IDENT: iden

traza.add(new Shift("iden",

                    new AtributoLexico[]{new
                    AtributoLexico(lex($1).leeId(),"lex",lex($1).leeValor())}));
```

Aunque el símbolo terminal + no tenga asociado ningún atributo, debe de ser tratado de forma independiente, ya que necesitamos añadir el símbolo + al árbol de análisis sintáctico y por tanto el correspondiente comando Shift a la traza. Debemos dividir las producciones donde se encuentran este tipo de terminales en dos producciones separadas. A continuación tenemos un ejemplo:

**Exp -> Exp + Term** ⟶ { **Exp -> Exp MAS Term**, **MAS -> +** }

En la segunda de las producciones, se debe añadir el comando a la traza y para ello asociamos a la producción la siguiente acción:

```
traza.add(new Shift("+", new AtributoLexico[]{}));
```

La lista de atributos léxicos estará vacía, porque como hemos dicho anteriormente, este terminal no tiene asociado ningún atributo.

### 5.4.1.3 Comando Valor

Este comando será añadido a la traza cuando se conozca el valor de alguno de los atributos asociados a nodos cuyo valor era desconocido hasta el momento.

*Valor (IdentificadorAtributo, ValorResultado)*

- *IdentificadorAtributo* es el identificador numerico único que se ha asociado con el atributo de un símbolo cuando éste ha sido creado.
- *ValorResultado* es el valor recién calculado del atributo indicado en el primer parámetro que constituye el comando.

Estos comandos serán añadidos a la traza cuando todos los atributos de los que depende otro, han sido calculados y por tanto el valor de este último puede ser calculado. El método del que hacemos uso para la incorporación de este tipo de comandos a la traza es **ponValor(valor,traza)**. Obsérvese que, en la clase Atributo, el método ponValor está sobrecargado para no admitir o admitir la traza. Este método tiene como misión, además de la incorporación del comando Valor, establecer internamente que un determinado atributo ha sido calculado y por lo tanto esto tiene que ser notificado en las variables que podrían depender de ese atributo. De esta forma, si un atributo tiene calculados los valores de los atributos de los que depende excepto este último o bien solo depende de él, será el momento en el que se ejecutará la acción asociada a ese atributo que dependía del que acabábamos de calcular. Gracias a esto, podemos obtener una secuencia de evaluaciones que serán las encargadas de mostrar finalmente en el árbol de dependencias el orden progresivo en el que los valores de los atributos son calculados. A continuación, se encuentra un ejemplo.

$Exp_0.val = Exp_1.val + Term.val$

```
valOfExp0.eq(new Atributo[]{valOfExp1,valOfTerm},
new Accion() {
        public void ejecuta() {
            valOfExp0.ponValor(valOfExp1.leeValor()+valOfTerm.leeValor(),traza);
}});
```

### *5.4.1.4 Comando Dep*

Este comando representa el establecimiento de las dependencias entre atributos de dos nodos.

*Dep (AtributosRequeridos, AtributosDependientes)*

- *AtributosRequeridos* representa los atributos necesarios para el cálculo de los atributos dados por el segundo parámetro.
- *AtributosDependientes* representa los atributos que están a la espera de ser calculados porque dependen de los valores de otros atributos cuyo valor es posible que no sea conocido hasta el momento.

Un comando de tipo Dep es añadido a la traza en cada una de las ecuaciones semánticas que muestran una relación de dependencia entre atributos asociados a símbolos de la gramática.

Dada la producción *Exp -> Exp + Term* de la gramática de la Figura 5.3.2, a continuación se muestra cómo se añaden los comandos a la traza para cada una de las ecuaciones semánticas pertenecientes a dicha producción.

$Exp_1$.varsh = $Exp_0$.varsh y Term.varsh = $Exp_0$.varsh

```
traza.add(
    new Dep(
        new String[]{varshOfExp0.leeId()},
        new String[]{varshOfExp1.leeId(), varshOfTerm.leeId()}
        )
    );
```

$Exp_1$.val = $Exp_0$.val + Term.val

```
traza.add(
    new Dep(
        new String[]{valOfExp1.leeId(),valOfTerm.leeId()},
        new String[]{valOfExp0.leeId()}
        )
    );
```

# 6. CONCLUSIONES Y TRABAJO FUTURO

## 6.1 Conclusiones

En este trabajo se ha desarrollado un entorno gráfico de depuración para gramáticas de atributos. El trabajo realizado ha satisfecho los objetivos planteados al comienzo de su realización.

Se han adquirido conocimientos sobre tecnologías que eran desconocidas hasta el momento. Se ha profundidado en los conocimientos adquiridos mediante la asignatura de Procesadores del Lenguaje y se han puesto en práctica los ya existentes.

Se ha comprobado la utilidad de la herramienta de depuración mediante la costrucción de una lista de comandos a través de la herramienta BYacc, entendiendo como ésta funciona y cómo podemos utilizarla para codificar gramáticas de atributos, así como para instrumentar dichas gramáticas para permitir la depuración con BYacc.

## 6.2 Trabajo futuro

La herramienta de depuración contruida puede ser mejorada y ampliada de diversas formas. En concreto, como líneas de trabajo inmediato proponemos:

- Añadido de la opción de incorporación de un archivo de configuración XML.
- Mejoras en el dibujado del árbol para gramáticas más complejas y extensas.
- Creación de una herramienta que traduzca automáticamente gramáticas de atributos en codificaciones BYacc, y que permita instrumentar automáticamente las mismas para permitir su depuración con ADebug.

# 7. INDICE DE FIGURAS

# 8. REFERENCIAS


1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (second edition). Addison-Wesley. 2007

2. Bennett, J.P. Introduction to Compiling Techniques: A First Course using ANSI C, LEX and YACC. McGraw Hill. 1990

3. Levine, J.R., Mason T., Brown Doug. Lex & Yacc. O'REILLY. 1995

4. Ablas, H. Attribute Evaluation Methods. En Ablas, H., Melinchar, B (eds.) "Attribute Grammars, Application and Systems". Lecture Notes in Computer Science 545. Springer. 1991.

5. Grosjean, J., Piccolo's Documentation. Disponible en http://www.cs.umd.edu/hcil/jazz/index.shtml. 2005

6. Bederson, B.M., Grosjean, J., Meyer, J., Toolkit Desing for Interactive Structured Graphics, *IEEE Transactions on Software Engineering,* 30 (8), 535-546. 2004

7. Hurka T.BYacc's Manual. Disponible en http://byaccj.sourceforge.net/. 2008

8. Johnson, S.C. YACC: Yet Another Compiler-Compiler. Unix Programmer's Mannual Vol 2b, 1979.

9. Piroumian V. Java GUI Development.1999

10. Sánchez, A., Huecas, G., Fernández Manjón, B., Moreno, P. Java 2: iniciación y ç Referencia. Ed. McGraw-Hill. 2001

11. Sarasa, A., Sierra, J.L. Fernández-Valmayor, A. Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de E-Learning. IEEE RITA, *en prensa*. 2009b

12. Hudson, S.E. CUP User's Manual. Disponible on-line http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html. 1999

13. Gosling, J., Joy, B., Steele, G.L., Bracha, G. The Java Language Specification Third Edition.Addison Wesley. 2005

14. Martinez, A., Temprado, B., "XLOP - XML Language-Oriented Processing" (Proyecto fin de carrera, Universidad Complutense, 2009).

15. Appel, A.W. Modern Compiler Implementation in Java. Cambridge Univ. Press

16. Knuth, D.E. Semantic of Context-free Languages, MathSystem Theory 2(2), 127-145, 1968. See also Math.System Theory 5(1), 95-96. 1971

17. Eckel, B. Thinking in Java. Prentice-Hall. 2002